Homework 1 – Supervised Algorithms

Objective is to explore multiple supervised algorithms on an interesting data problems

Problem

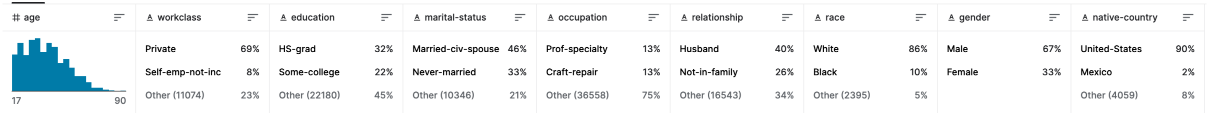
Based on the data collected about individual salaries in the “Census Income” dataset in 1994, we will employ multiple learning algorithms to classify an individual as “rich” or “not”, with the definition of “rich” being income >= 50K, typical Boolean classification problem.

Data

The dataset contains 48842 records of individual salaries with the following features

Feature name	Description
Employment	Employed in private corporation, government or self-employed
Education	Level of education of the individual. High school, Bachelors, Masters etc
Marital Status	Whether the individual has been married, single or divorced
Position	Profession capacity of the Individual. Clerical, Managerial, Exec, etc
Family Status	Status in the family. Husband, Wife, Not in family etc
Ethnicity	Ethnicity of the individual
Sex	Sex of the individual
Country	Country of the individual
Salary	Captures if the income is >50K or <= 50K

Rough data distribution (Fig 1)



# age	workclass	education	marital-status	occupation	relationship	race	gender	native-country
	Private 69%	HS-grad 32%	Married-civ-spouse 46%	Prof-specialty 13%	Husband 40%	White 86%	Male 67%	United-States 90%
	Self-emp-not-inc 8%	Some-college 22%	Never-married 33%	Craft-repair 13%	Not-in-family 26%	Black 10%	Female 33%	Mexico 2%
17 90	Other (11074) 23%	Other (22180) 45%	Other (10346) 21%	Other (36558) 75%	Other (16543) 34%	Other (2395) 5%		Other (4059) 8%

(Source: https://www.kaggle.com/datasets/wenruliu/adult-income-dataset)

We will analyze and understand individual algorithms, explore and optimize classifier parameters and finally we will compare algorithms with each other

For each of these algorithm, we will do the following in sequence

- Finding the right split of testing and training data
- Optimize classifier parameters for generalization and runtime performance

Optimizing splits

It is important to come up with a good criteria for choosing how much of the dataset are we planning to train on and how much we allocate for testing. We need to be able to train on a good representation of the real-world data so that the learned model can perform accurate prediction, but we test the generality of the learned model by testing on the test data. So balancing the size and variety in both splits is important. We will pick the most accurate default classifier from the pre-determined split value set of [0.2, 0.25, 0.3, 0.35, 0.4, 0.5]. This classifier is also cross-validated with 5 folds on the training data and accuracy is the mean of all the fold runs and the test data. This is done to make the accuracy metric more reliable and general. We will not look at training or testing performance as we will deal with that in the next section.

Optimize classifier parameters

As with all learning algorithms, it is easy to tune an algorithm to work really well for the training and testing dataset but does not always guarantee generalized accuracy in the real-world. This is called "over-fitting". Overfitting can result in inaccurate results due to lack of generalization and/or very in-efficient algorithms that can be very costly to repeatedly train and predict in the real world. Balancing cost and accuracy is an important trade off when tuning the input parameters to a classification algorithm. We will use a technique called "Cross-validation" to pick a chosen subset of these parameters to avoid over-fitting.

Note that both these considerations are not independent, for e.g. choosing the classifier parameters can have impact on the data splitting and vice-versa. For simplicity we will do them in an order and assume that the impact is minimal. This is just an assumption for simplicity.

Picking the most optimum classifier parameters, requires us to iterate over the possible value space for each of these parameters, train the model and assign a **classifier score** based on the following 3 factors

- Cross validated accuracy score
- Normalized time taken to train
- Normalized time taken to test

Note that typically in production environments exhaustive search for optimizing the parameters is very important. ML Libraries may provide helper functions to search over the parameter space and find optimum values. (scikit has GridSearchCV) but for simplicity and the need to factor in training and testing time in addition to accuracy, we do not do an exhaustive search but stick to picking from discrete value sets as long as the accuracy is acceptable.

The weighted formula for calculating the classifier score is

```
W_accuracy * classifier_accuracy +
W_training_time * (1 - (Train_current) / Train_max)) +
W_testing_time * (1 - (Test_current / Test_max))
```

- Where *w_accuracy, W_training_time, W_testing_time* are fraction weights assigned to how important is accuracy, time taken to train, time taken to test and they all add up to 1 such the total score is a fraction as well.
- *Train_max* is the maximum time taken to train for all possible configuration and *Train_current* is the time taken to train in the current configuration
- *Test_max* is the maximum time taken to test across all possible configurations and *Test_current* is the time taken to test in the current configuration

Note that ideally we include the 'cost' for training and testing, which quantifies the hardware resources required (CPU/memory) as well as time into the classifier score formula.

In a perfectly scalable setup (cloud), time does not matter as we can potentially train and predict in constant time. For the sake of simplicity, we just stick to time here.

Each algorithm, based on the internals on how the training is done and the data structure used to retain information used to predict, will come up with weights and compute the classifier scores. Eventually the classifier with the highest classifier score is picked as the winner and tested on.

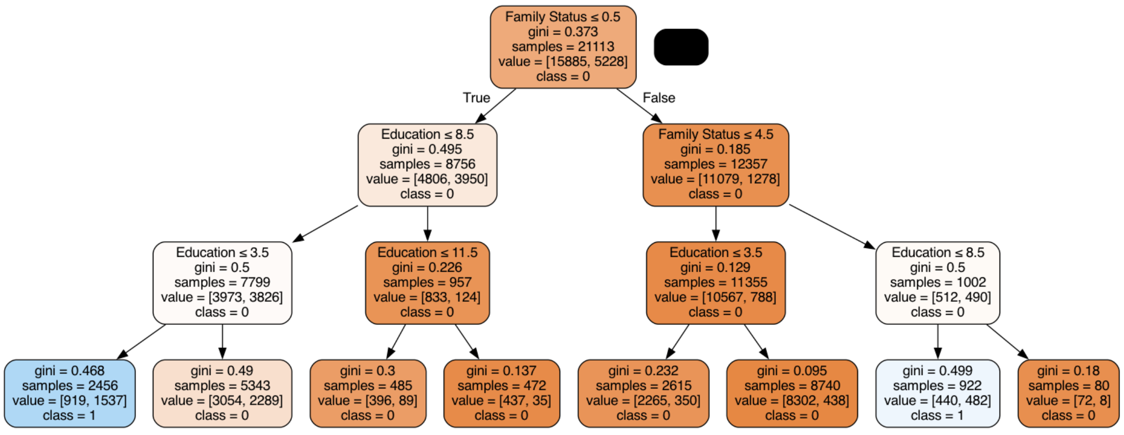
DECISION TREES

Intuitively by scanning at the data distribution (Fig 1) of the columns in the data,

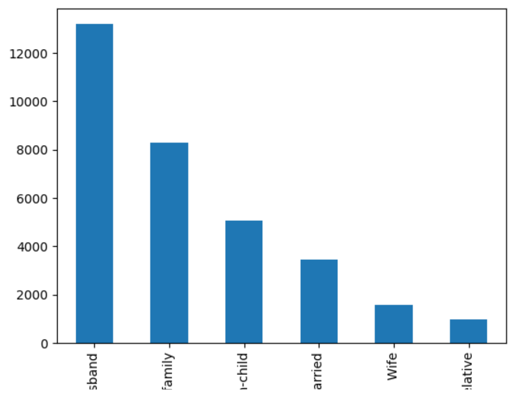
- Features like *Family status, Marital status and Education* seems to have a good split on the data, hence could be good candidates for root node of the tree
- Features like *Sex, Country and Employment* are heavily skewed on a single value that splitting on this feature would not be very beneficial being the root node.

Note that we don't know the data distribution after the initial split, hence the same assumption on column distribution does not hold.

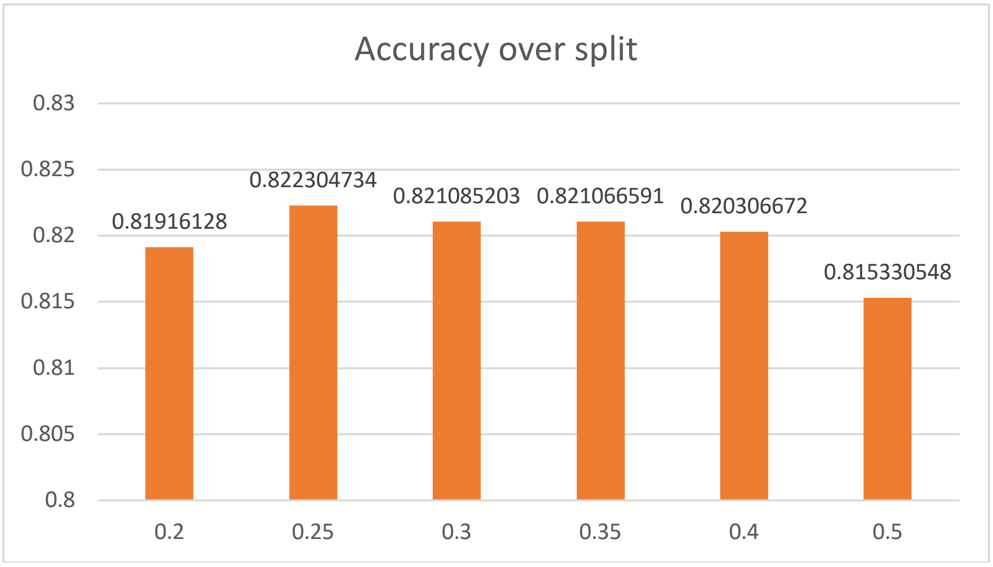
Visualizing the tree (with max depth=3) confirms that



Family column distribution roughly splits the data into half between the values 'Husband' (12357) and others (8756) which is shown in the root node of the decision tree.



Splitting testing and training data



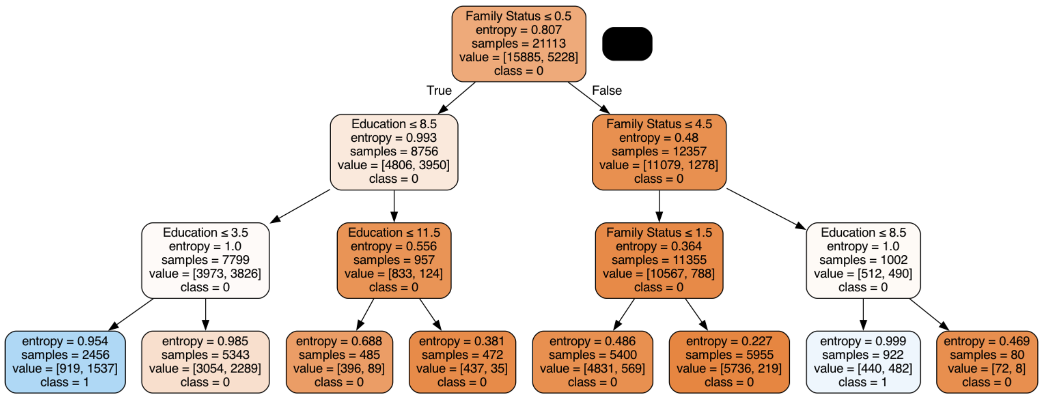
Chosen split: 0.25

Tuning classifier parameters

Decision trees can get quite deep and unnecessarily complex, so pruning the tree branches is an important optimization for performance and explain-ability.

Classifier property	Description	Default value
Classification criteria	Heuristic for selecting the splitting criteria that creates the best possible partition for optimum performance	Gini (Other values could be 'entropy')
Max depth	Pruning parameter to limit the depth of the tree. Limiting the depth improves performance and explain-ability.	None
Min samples leaf	Ensuring that a leaf node has a minimum number of samples ensures nodes of lesser value is folded reducing the complexity of the tree	1

Classification criteria:
'Gini' index measures the probability of a random instance being misclassified while the 'entropy' classification criteria measures the impurity (instances that don't belong) in each classification. In our dataset, we are dealing with features which are very low-dimension and really clear choices for top 3 levels of the tree. This is shown in the figure below with 'entropy' classification criteria. We may see one criteria performing better than the other in high dimensional feature distribution but for our training set, we can just stick to the default 'Gini' index. Visualizing 'entropy' tree confirms our assumption.



Given the time constraints and to simplify by reducing the optimization space, we have picked a reasonable value set

- max_depth=[3,5,10]
- min_samples_leaf=[10,100])

We will compute the classifier score for all combinations of the parameter values and we will pick the best classified parameters.

The weights in the classifier score formula (as mentioned before) is as follows

- W_accuracy ➔ 0.7 (Training dataset accuracy is most important)
- W_training_time ➔0.2 (Construction of the tree and pruning of the trees based on min samples leaf could be costly)
- W_testing_time ➔0.1 (Tree pruned sufficiently and testing time should be negligible)

Results of the above trial is as follows

Max Depth	Min Samples leaf	Accuracy	Training time	Weighted Classifier score
3	10	0.78	0.0061	0.947
3	100	0.79	0.0061	0.949
5	10	0.81	0.0090	0.897
5	100	0.81	0.0090	0.896
10	10	0.82	0.0140	0.842
10	100	0.81	0.0125	0.852

Max depth = 3
Min Samples Leaf = 100

Observations

- By looking at the weighted score of accuracy and training time, we picked the configuration that may not overfit and is potentially orders of magnitude faster to train and predict. This will help us run learning more frequently on new data and prediction can be more real time as well

- By picking the more simpler decision tree with max-depth=3, the debuggability and explain-ability of the learned model is much higher which is critical for using learned models in production

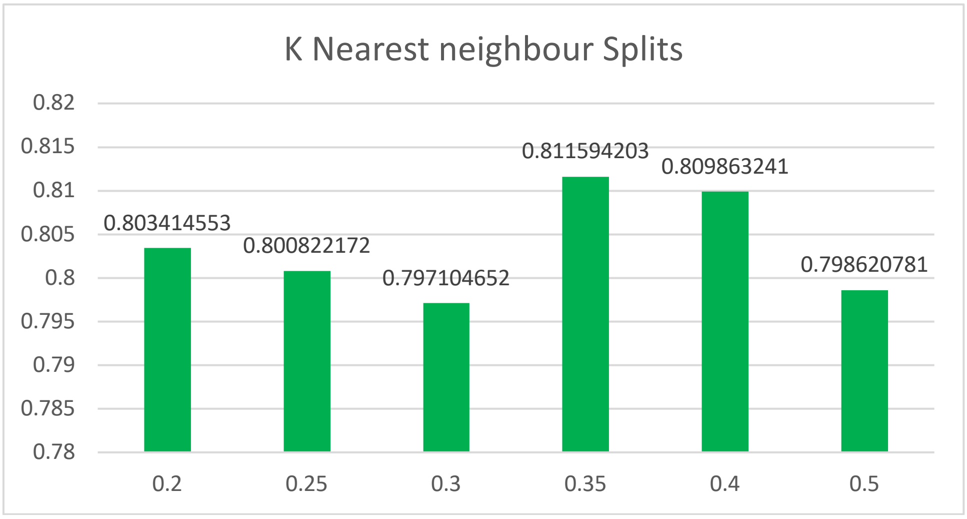
Testing results

With the parameters picked (max depth=3, min samples leaf=100), we finally test the prediction on the test data and we get the following accuracy which is close to what we saw in the cross-validated set as well.

accuracy: 0.790

K_NEAREST NEIGHBOURS

Splitting testing and training data



Chosen split: 0.35

It is critical to note the jump from 0.3 split to 0.35 split. One reason could be an internal switch between the algorithm. The default KNeighboursClassifier uses its own intelligence to determine the algorithm to use. It picks between brute-force, kd-tree and ball-tree based on the training data. I suppose sufficient deliberation has gone into if this is generic enough to be made as the default, so for this experiment we will trust that `auto` is reliable

Tuning classifier parameters

K Neighbours classifier has the following input parameters to tune for better generalization and run-time performance.

Decision tree classifier has the following input parameters that are significant to tune for better generalization and run-time performance. Tree can get quite deep and unnecessarily complex, so pruning the tree branches is an important optimization for performance and explain-ability.

Classifier property	Description	Default value
n_neighbors	Single most important heuristic to balance between prediction accuracy and runtime performance. K Neighbours is a lookup heavy algorithm as each prediction requires identifying K nearest neighbour in a N-dimensional matrix (n-features).	5
leaf_size	For small enough data size, brute force can be more efficient than tree-based. The tree based algorithms will fallback to brute force in the leaf nodes determined by the value of this parameter. This can help cut down the size of the tree being built which can help with training and testing time.	30

We have picked a reasonable value set

- **n_neighbours**=[3,5,10]
- **leaf_size**=[30,100,500])

The weights in the classifier score formula (as mentioned before) is as follows

- W_accuracy ➔ 0.6 (Training dataset accuracy is most important)
- W_training_time ➔0.1 (Construction of the tree based structures to help with finding neighbours can be costly, but a high leaf_size helps reduce the construction time)
- W_testing_time ➔0.3 (Finding K neighbours in runtime can be costly even with the help of tree based structures)

Results of the above trial is as follows

N Neighbours	Leaf size	Accuracy	Training time	Testing Time	Weighted Classifier score
3	30	0.78	0.0055	0.217	0.579
3	100	0.79	0.0042	0.229	0.596
3	500	0.79	0.0033	0.316	0.543
5	**30**	**0.80**	**0.0050**	**0.214**	**0.601**
5	100	0.80	0.0041	0.242	0.594
5	500	0.80	0.0031	0.326	0.546
10	30	0.81	0.0049	0.239	0.586
10	100	0.81	0.0039	0.271	0.578
10	500	0.81	0.0030	0.352	0.530

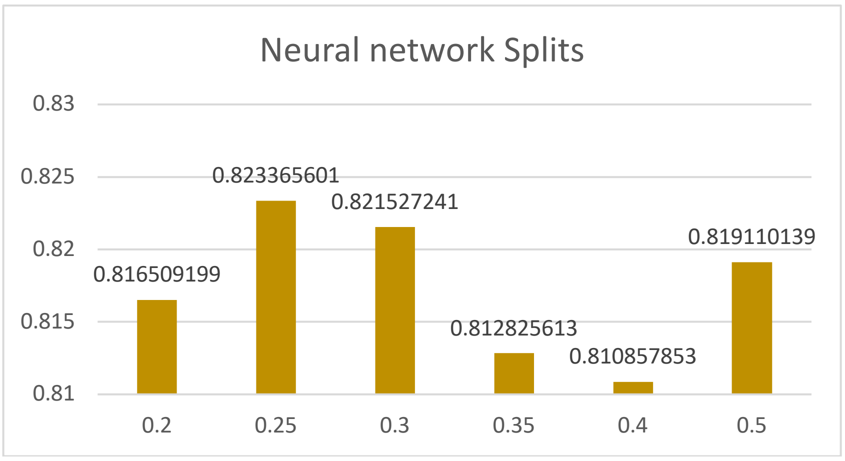
N Neighbours = 5
Leaf size = 30

Testing results
With the parameters picked, we finally test the prediction on the test data and we get the following accuracy which is close to what we saw in the cross-validated set as well.

accuracy: 0.811

NEURAL NETWORKS

Splitting testing and training data



Chosen split: 0.25

Tuning classifier parameters

Classifier property	Description	Default value
solver	Algorithm used for weight optimization	Adam (Other values include lbfgs, sgd). Adam is a good default that works for most datasets and converge quick.
hidden_layer_sizes	Defines the number of layers and the size of each layer. This is the most important heuristic to control the complexity of the network.	(100,) single hidden layer with 100 neurons.
alpha	Parameter used to avoid overfitting by penalizing weights with large magnitudes	0.0001

We are not sure if the data is linearly separable and hence it is critical to tune the hidden_layer_sizes. This is a 2 dimensional data and the possible space is huge. We need to follow rule of thumbs to come up with a set of possible values. We will pick one wide single layer (100,) and a narrow long layer(10,10,10).

For alpha we pick [0.01, 0.1, 1] as possible values.

The weights in the classifier score formula (as mentioned before) is as follows

- W_accuracy ➔ 0.6 (Training dataset accuracy is most important)
- W_training_time ➔0.3 (Construction of the network optimizing for the weights at each hidden layer)
- W_testing_time ➔0.1 (Negligible as we have network size sufficiently small)

Results of the above trial is as follows

Hidden_layer_sizes	alpha	Accuracy	Training time	Weighted Classifier score
(100,)	0.01	0.81	9.59	0.208
(100,)	0.1	0.80	7.01	0.334
(100,)	1.0	0.79	7.15	0.353
(10,10,10)	0.01	0.77	6.16	0.491
(10,10,10)	0.1	0.78	1.62	0.746
(10,10,10)	1.0	0.78	3.39	0.620

Hidden_layer_sizes=(10,10,10)
alpha=0.1

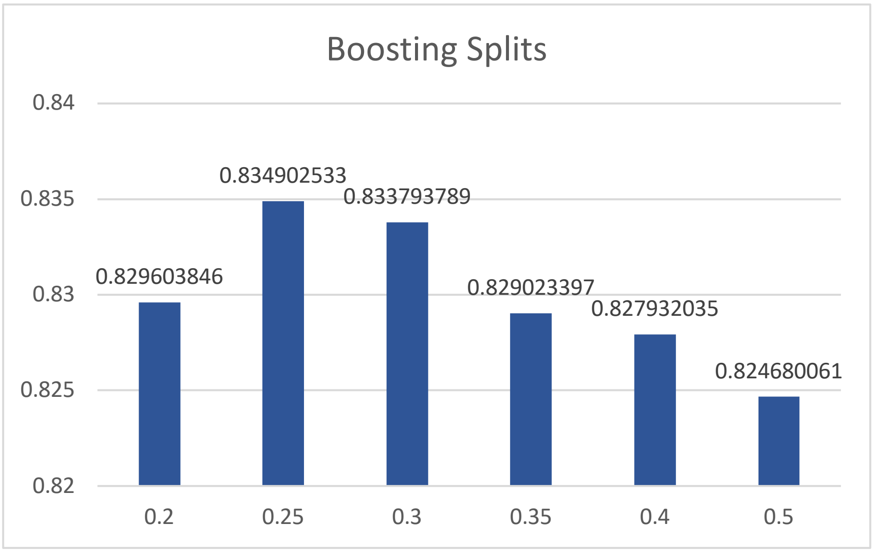
Testing results

With the parameters picked, we finally test the prediction on the test data and we get the following accuracy which is close to what we saw in the cross-validated set as well.

accuracy: 0.818

BOOSTING

Splitting testing and training data



Chosen split: 0.25

Tuning classifier parameters

The variation of the boosting algorithm we chose is the AdaBoostClassifier. The classifier has the following parameters to tune

Classifier property	Description	Default value
n_estimators	Number of estimators after which boosting is terminated. Number of iterations essentially.	50
Learning_rate	Weights applied to regressor at each boosting iterations. A higher learning rate increases the contribution of the each iteration in the final result.	1.0

We have picked a reasonable value set

- **n_estimators**=[1, 10, 100]
- **learning_rate** =[1, 10])

The weights in the classifier score formula (as mentioned before) is as follows

- W_accuracy ➔ 0.8 (Training dataset accuracy is most important)
- W_training_time ➔0.1 (Computing the distributions and next weak learners constribution is pretty efficient and hence does not require more than 0.1)
- W_testing_time ➔0.1 (testing is also fairly efficient and quick)

Results of the above trial is as follows

N Estimators	Learning Rate	Accuracy	Training time	Testing Time	Weighted Classifier score
1	1	0.75	0.0050	0.0008	0.798
1	10	0.75	0.0049	0.0008	0.798
10	**1**	**0.81**	**0.0455**	**0.0035**	**0.828**
10	**10**	**0.29**	**0.0459**	**0.0036**	**0.411**
100	1	0.82	0.4350	0.0298	0.696
100	10	0.75	0.4445	0.0318	0.633

N Estimators = 10
Learning rate = 1

Observations

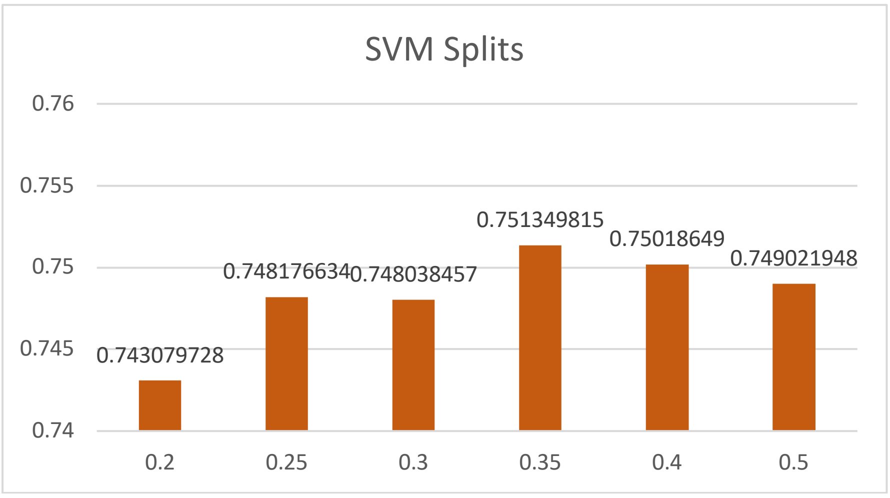
- In the line marked in RED in the above results, we observe worse accuracy because of what I think as underfit. The underlying algorithm used is SAMME.R. A really high learning rate assigned to subsequent iterations - focus too much on the 'hardest' data points (wrongly classified) with very iterations tend to not fit ALL the training data.
- For a given learning rate, more number of iterations the better the cross validated score as it not overfit like other algorithms

Testing results
With the parameters picked, we finally test the prediction on the test data and we get the following accuracy which is close to what we saw in the cross-validated set as well.

accuracy: 0.821

SUPPORT VECTOR MACHINES

Splitting testing and training data



Chosen split: 0.35

SVM effective in high dimensional spaces (hence not needed). number of dimensions is greater than the number of samples

Tuning classifier parameters

The variation of the boosting algorithm we chose is the AdaBoostClassifier. The classifier has the following parameters to tune

Classifier property	Description	Default value
kernel	Kernel transformation to be applied	rbf
C	trade-off between misclassifying training examples and the simplicity of the decision surface	1.0
Gamma	Defines how much influence a single training example has. This	scale

We have picked a reasonable value set

- **kernel** =['linear', 'rbf']
- **C** =[1, 5, 10]
- Gamma = We leave it as scale as this is dynamically adaptive

The weights in the classifier score formula (as mentioned before) is as follows

- W_accuracy ➔ 0.6 (Training dataset accuracy is most important)

- W_training_time ➔0.3 (Training time for SVM is pretty intensive, we will factor in with a weight of 0.3 into the score)
- W_testing_time ➔0.1 (testing is also fairly efficient and quick)

Results of the above trial is as follows

Kernel	C	Accuracy	Training time	Testing Time	Weighted Classifier score
Linear	1	0.75	3.26	0.797	0.754
Linear	5	0.75	8.64	0.799	0.638
Linear	10	0.75	13.36	1.22	0.523
RBF	1	0.75	3.44	2.921	0.679
RBF	5	0.75	4.59	2.899	0.655
RBF	10	0.75	4.101	2.87	0.666

Kernel = linear
C = 1

Observations

- SVM performs well in high dimensional spaces and the dataset we work on contains 8 features and cardinality of each of these features is low. SVM is probably an overkill as shown by the training time required to build the model

Testing results
accuracy: 0.751

Comparison of all algorithms

For this simplistic dataset we chose, simple rule based Decision tree performed really well in terms of runtime performance with good accuracy mainly because there were clear choices for the root and the top few nodes.

Boosting did fairly well as well. Other algorithms like K Nearest neighbours assume that all features are equally weighted and finding nearest neighbours during prediction was not cheap and neural networks optimization space was huge and is probably an overkill to go through the exercise of finding the right number of layers and maximum neurons in each layer. SVM did not perform well in runtime performance or accuracy as the dataset is not a good fit in terms of high dimensionality requirement.