Machine Learning Coursework

Patrick Lee
an20421

1 Analysing fashion-MNIST

The aim of this section is to analyse the fashion-MNIST dataset using unsupervised learning methods - namely K-Means clustering and Gaussian Mixture Models. Unsupervised learning is widely used to find "interesting structure" within the data without feeding the model the desired output for each input. The fashion-MNIST dataset consists of 28x28 grayscale images and an associated label from 10 classes (each representing a type of clothing), giving a total of 784 input variables (each a number between 0 and 255 representing pixel colour). The large number of input variables introduces the "curse of dimensionality", requiring an exponentially large number of training data to account for the majority of input permutations, in this case in the order of n^{784}.

In order to solve this problem, the dataset will be normalized (by dividing all data-points by 255) and then reduced down to **two dimensions** by Principle Component analysis.

1.1 Principle Component Analysis

Many datasets can be approximated by removing variables which account for small proportion of the variance within the dataspace. PCA is one such technique for acheiving this. In this section, the aim is to reduce the dimensionality of the Fashion-MNIST dataset by projecting the data onto the two vectors which result in a maximised variance.

By differentiating the variance of the projected data and setting it to zero, we get $Su_1 = \lambda_1 u_1$ where S represents the covariance matrix of the dataset, u_1 is an eigenvector representing the first principle component and λ_1 is its corresponding eigenvalue. As all principle components must be orthogonal, the second component u_2 is the next direction which maximises the projected variance of the data (has the next largest corresponding eigenvalue), subject to being orthogonal to u_1

The proportion of the total variance of the data which is accounted for by the first n principle components is given by

$$\frac{\sum_{i=1}^{n} \lambda_i}{\sum_{i=1}^{N} \lambda_i}$$

where λ_i represents the ith eigenvalue in the sorted list of components.

By fitting Scikit Learn's PCA model to the fashion-MNSIT dataset, keeping two principal components, a proportion of **0.468** was given, showing that the colour of just 2 pixels of a 28x28 image accounts for nearly half variance of the total dataset.

Once the data was projected onto the first two principle components, it was plotted as a scatter graph as shown in **Figure 1** with colour use to show different classes. It can be seen the points generally cluster visibily into their respective classes. The average distance from each point to the mean x value of a given class was found to be **2.86** and the average distance between these means was found to be **6.12**,

showing that on average, points are closer to their cluster means that that of other clusters. However, the average distance between a cluster mean and that of its nearest cluster was found to be **1.64** showing that, on average, points are closer to a neighbouring cluster than their own.

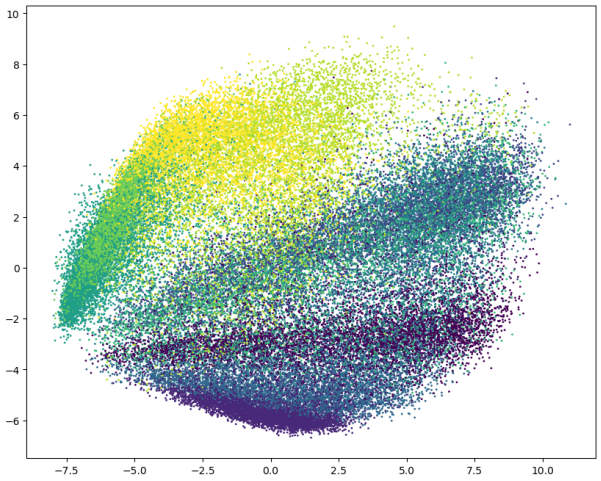

Figure 1

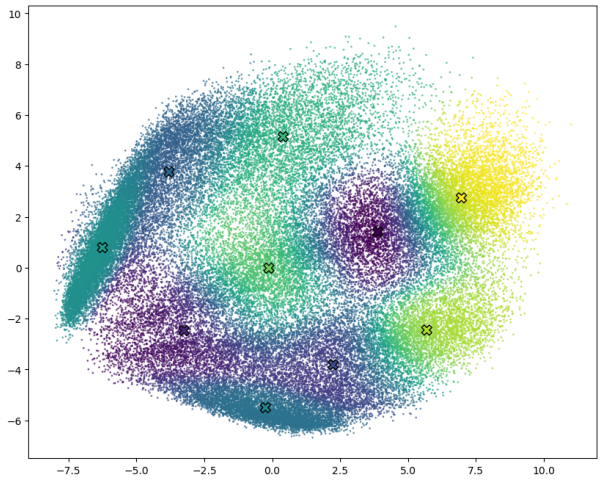

Figure 2

1.2 Gaussian Mixture Models

Hard clustering methods such as K-means have the disadvantage of not accounting for the probabilities of a data-point being part of other clusters. A Gaussian mixture model can be used to address this by creating linear superposition of Gaussian distributions, each one corresponding to one of K output classes as shown below.

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \Sigma_k)$$

Each Gaussian has a mixing coefficient π_k which is associated with a K dimensional latent variable $\mathbf{z}$, in which a single $z_k = 1$ and all other $z_i = 0$ and $p(z_k = 1) = \pi_k$. This latent variable is combined with $\mathbf{x}$ to give the joint distribution $p(\mathbf{x}, \mathbf{z})$.

Creating this model, requires us to find π_k, μ_k and Σ_k for each of the K classes. In this case, the Expectation Maximisation algorithm is used to find these parameters by increasingly refining them until either the likelihood function is maximised with respect to these parameters or the parameters themselves converge. After being assigned initial values, each iteration of the algorithm sees the responsibilities of the current parameter values calculated and then used to calculate new parameter values. These responsibilities make use of the join distribution $p(\mathbf{x}, \mathbf{z})$ and are represented as $\gamma(z_k) \equiv p(z_k = 1|\mathbf{x})$ i.e. the responsibility component k takes for explaining the observation x. The log-likelihood is then calculated using these parameters and the algorithm stops if the result of this has converged.

SciKit's GaussianMixture model, which uses this EM algorithm for optimization, was fitted to the fashion-MNIST data (as transformed before in **1.1**) and the resulting clusters plotted in **Figure 2** with the colour blend of each point representing the probabilities of it being drawn from each of the 10 Gaussian distributions, the means of which are also plotted.

Two metrics were used to assess the quality of clustering, namely the Completeness score (proportion of data points of the same class which are also in the same cluster) and Homogeneity score

(proportion of clusters which contain only data points of one class). These scores were **0.420** and **0.417** respectively, which align with the fact that roughly half of the clusters in the plot appear to match well-defined gaussians.

2 Classifiers

This section aims to compare the performance of two classifiers on the fashion-MNIST dataset, namely Artificial Neural Networks and Support Vector Machines

2.1 Artificial Neural Networks

Here the aim is to analyse the results of using an artificial neural network to classify points within the fashion-MNIST dataset. Here, a multilayer perceptron network is used which is an example of a feed-forward network (differing from the more complex recurrent networks in that signals can only travel from input to output), using Backpropagation as its method of supervised learning.

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right)$$

The equation above illustrates a multilayer perceptron with a single hidden layer of M neurons, mapping D inputs to K outputs. For each of the hidden neurons, the non-linear activation function h is applied to the weighted sum of all input neurons which are connected to it. The result of this is passed towards each connected output neuron, which will apply the sigmoid function to the weighted sum of all activation results of hidden neurons which are connected to it. The biases parameters $w_{j0}^{(1)}$ and $w_{k0}^{(2)}$ are also included in these summations. In this case, there are 10 outputs, each corresponding to one of the classes in the fashion-MNIST, with the result of the sigmoid function representing the estimated probability of the given input data being part to this class.

During learning, the calculation of the outputs is followed by use of Backpropagation to optimise the weights. Neural networks are effective methods for non-linear classification problems such as this, although their flexibility often results in a high risk of over-fitting if the testing set is small.

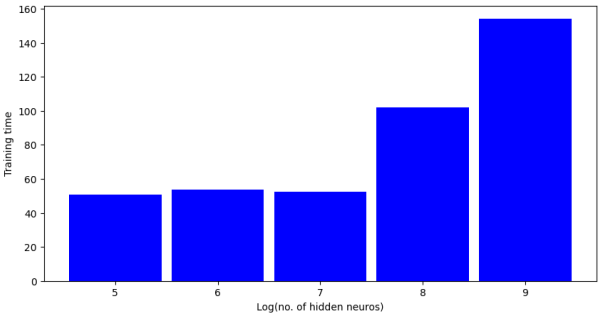

Figure 3

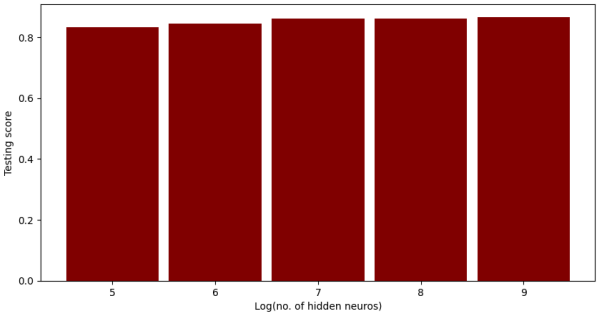

Figure 4

I initially trained neural networks with a range of hidden layer sizes to find good a trade-off between performance and training time. Using sizes between 2^5 and 2^9, which covers a large range of suitable sizes between the 785 inputs and the 10 outputs, the plots of hidden layer sizes against time and testing score are displayed in **Figure 3** and **Figure 4**. From this, it can be seen that increase in training time significantly exceeds the score improvement at hidden layer sizes more than 2^7, which was ultimately used as the hidden layer size M.

The neural network was trained using 10,000 samples from the fashion-MNIST with the learning and validation curves plotted below in **Figure 5**. The distance between the learning and validation

curves shows that the model is over-fitting to the training data. The training curve appears to start moving downwards (whilst the validation curve continues upwards) towards the end when a large number of samples have been used, which ties in with the fact a larger set of training data tends to reduce over-fitting and allow for better predictions of unseen data.

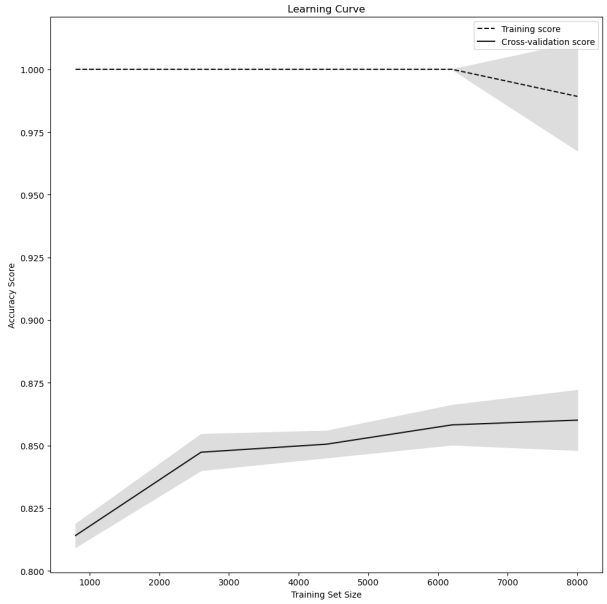

Figure 5

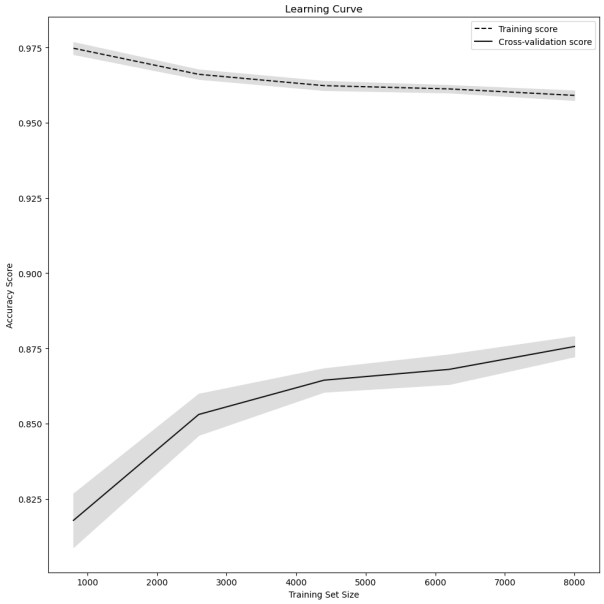

Figure 6

When tested on a different subset of 10,000 samples, the model gave a mean accuracy of **86.00%**. Whilst fairly accurate, the learning curve and average training score of **100%** show that the model has over-fit. This could be improved by using more data (which as the disadvantage of requiring more time to train) and tuning the hyper parameters. Using a deep learning approach with more hidden layers also may well improve accuracy but would likely over-fit more due to the difficulty of keeping generality of the more complex decision boundaries unless a significantly larger data-set was used.

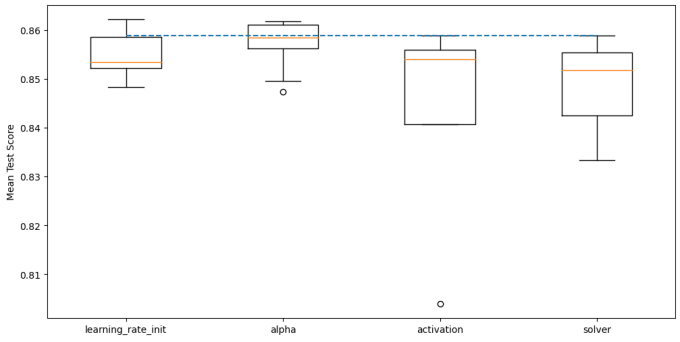

Figure 7

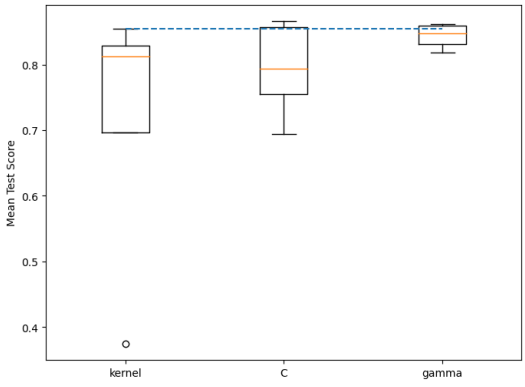

Figure 8

The multi-layer perceptron model has a large number of different hyper-parameter which can effect performance with different types of data-sets. As good size for the hidden layer was found previously, I focused on the following parameters from SciKit Learn to tune for the model:

- Activation function - Identity, Logistic Sigmoid, Hyperbolic Tan Function or Rectified Linear Unit Function.
- Solver for weight optimisation - Stochastic Gradient Descent (standard and Adam version), Limited-memory BFGS.

- Alpha (coefficient of the regularization term used in the cost function) - used a 100 values between $\log_{10} -3$ and $\log_{10} 0$.
- Initial learning rate (used as a constant)- used 100 values between $\log_{10} -4$ and $\log_{10} 2$.

SciKit Learn's RandomizedSearch model was used to fix and validate the neural network with a variety of values for these parameters. The box plot in **Figure 7** shows how these parameters effect the validation score of the configuration, with the blue dotted line showing the score of the model with these parameters at their default settings. It can be seen that in the case of the Activation Function and Solver parameters, their default values of Rectified Linear Unit Function and the Stochastic Gradient Descent (Adam version) give the best score. The 'Alpha' and 'Initial Learning Rate' parameters were further tuned using a narrower range about the previously found values. These optimal values were found to be **0.02535** and **0.00132** respectively.

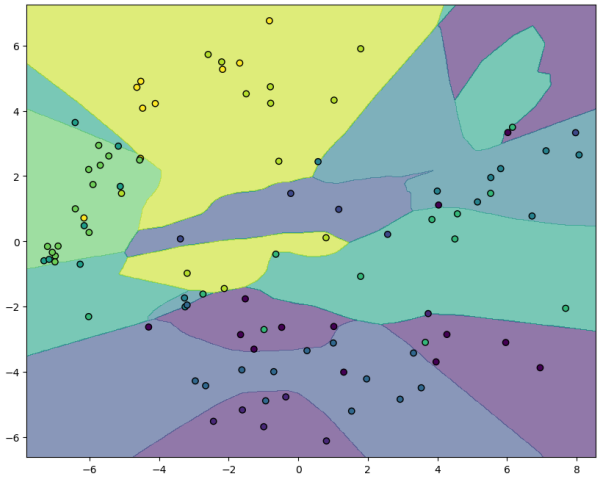

Figure 9

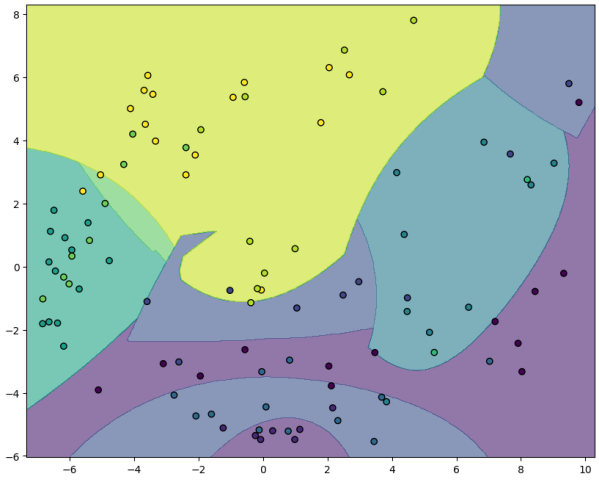

Figure 10

For the purposes of displaying the data, a smaller set of 100 samples was taken from fashion-MNIST and projected onto two dimensions as described in **Section 1.1**, the Neural Network was trained with this data again with the same configuration of hyper-parameters and the decision boundaries were plotted as shown in **Figure 9** with colours representing the true class label of the points and the decision boundaries from prediction on the training set.

These plots do a good job of showing how the flexibility of neural netowrks and how their decision boundaries contrast with those of other models (such as the more elliptic ones of Gaussian Mixture Models). In this case, the use of a reduced sample size likely means that the model is more over-fitted and the decision boundaries more complex than they would otherwise be.

2.2 Support Vector Machines

Support vector machines are models which use a given kernel to construct a hyper-plane to separate classes by maximising the distance between such a plane and the nearest training data points for any class. support vector machines make use of 'the kernel trick'

I used SciKit Learn's Support Vector Classifer, setting the regularization parameter to **4.0**. The model was then trained it using 10,000 samples from the fashion-MNIST with the learning and validation curves plotted in **Figure 6**. The distance between the learning and validation curves shows that the model is over-fitting to the training data. The training curve appears to start moving downwards (whilst the validation curve continues upwards) towards the end when a large number of samples have been used, which ties in with the fact a larger set of training data tends to reduce over-fitting and allow for better predictions of unseen data.

When tested on a different subset of 10,000 samples, the model gave a mean accuracy of **86.75%**, showing a slight increase over that of the Neural Network. Whilst fairly accurate, the learning curve and average training score of **100%** show that the model has over-fit. This could be improved in a similar fashion to the neural network.

For tuning the model, the following hyper-parameters were focussed on:

- Kernel function used in the model - 'linear', 'poly', 'rbf' or 'sigmoid'.
- Regularization parameter. - used a 100 values between $\log_{10} -2$ and $\log_{10} 1$.
- Gamma (kernel coeffecient for 'rbf', 'poly' and 'sigmoid' kernels) - used a 100 values between $\log_{10} -3$ and $\log_{10} -1$.

The RandomizedSearch model was again used to fit and validate the neural network with a variety of values for these parameters. **Figure 8** shows how these parameters effect the validation, plotted in the same format as with the neural network. It can be seen that the Kernel function for the model produces optimal results at its default setting of 'rbf' and that the tuning the regularization parameter and kernel coeffecient can have a slight positive effect on perfmormance These optimal values were found to be **8.11** and **0.0236** respectively.

Figures 10 and 11 show how both training time and test scoring compare between the Neural Network and Support Vector Classifer. It can be seen from this that the latter performs slightly better in scoring and significantly better in terms of training time. It could be assumed from this that the SVM is the better model, especially if training time is a critical factor. However, the ability to add a number of hidden layers to neural networks means gives them more flexibility.

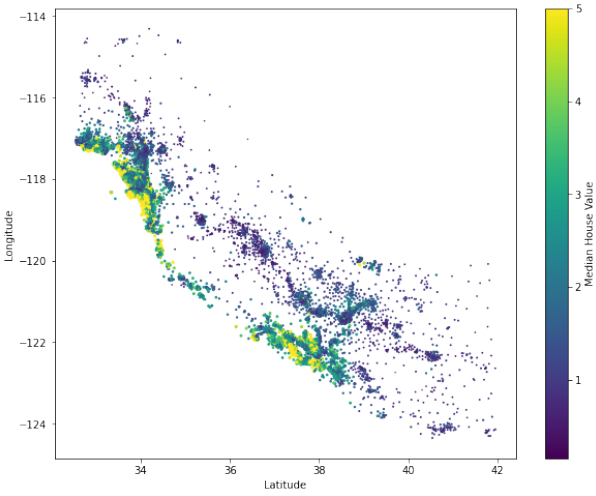

Figure 9

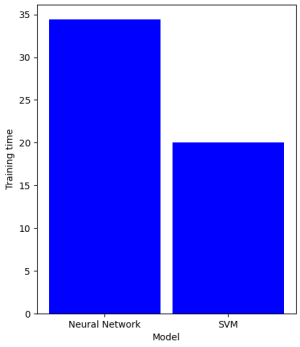

Figure 10

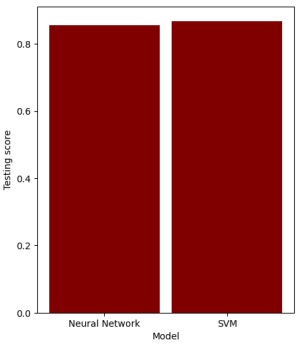

Figure 11

3 Bayesian linear regression with PyMC

The aim of this section is create a model which can predict 'Median House Value' in the california data-set using Bayesian linear regression. As a whole, the aim of linear regression is to compute weights **w** such that predictions can be made from **x** using $\mathbf{y} = \mathbf{wx}$. For complex data-sets, the ability to make predictions and exact inferences beyond the scope of the data becomes intractable. A Bayesian approach allows approximation of the model parameters, in this case **w**, over a distribution conditioned on the observed data given by $P(\theta|D = d)$. We can compute an approximation of **w** by sampling from $P(\mathbf{w}|D = d)$ and then taking the mean.

The scatter plot 'latitude' against 'longitude' is shown in **Figure 9**, with the colour indicating

'median house value' (purple to yellow corresponding with low to high). It can be seen that 'longitude' and 'latitude' are negatively correlated with the graph roughly mirroring the shape of the state of California. With the more expensive houses dominating the left edge of this shape, it can be inferred that proximity to the coast has a positive effect on median house price, with the major cities accounting for the larger clusters of these higher value properties. The plot also shows that given a higher latitude, a lower longitude gives a higher house value and vice versa.

The data was transformed using SciKit's Standard Scaler which removes the the mean and scales the each feature vector x_i down to unit variance - given by $z_i = \frac{x_i - \mu_i}{\sigma_i}$. This ensures all features are a common scale and often makes machine learning algorithms faster to run. I decided that it wasn't necessary to clean the dataset as there appears to be few significant outliers and no null/invalid points in the data-set used.

For each weight, a prior distribution of $\mathcal{N}(0, 20)$ was used - the high variance representing the lack of initial knowledge of the true distributions. PyMC was then used to obtain posterior distributions for each model parameter using the Hamiltonian Markov Chain Monte Carlo. This MCMC works by sampling from a sequence of marginal distributions and only keeping samples once the distributions become similar to $P(\theta|D = d)$ as mentioned previously. The resutling posteriors for each parameter are shown in **Figure 12**

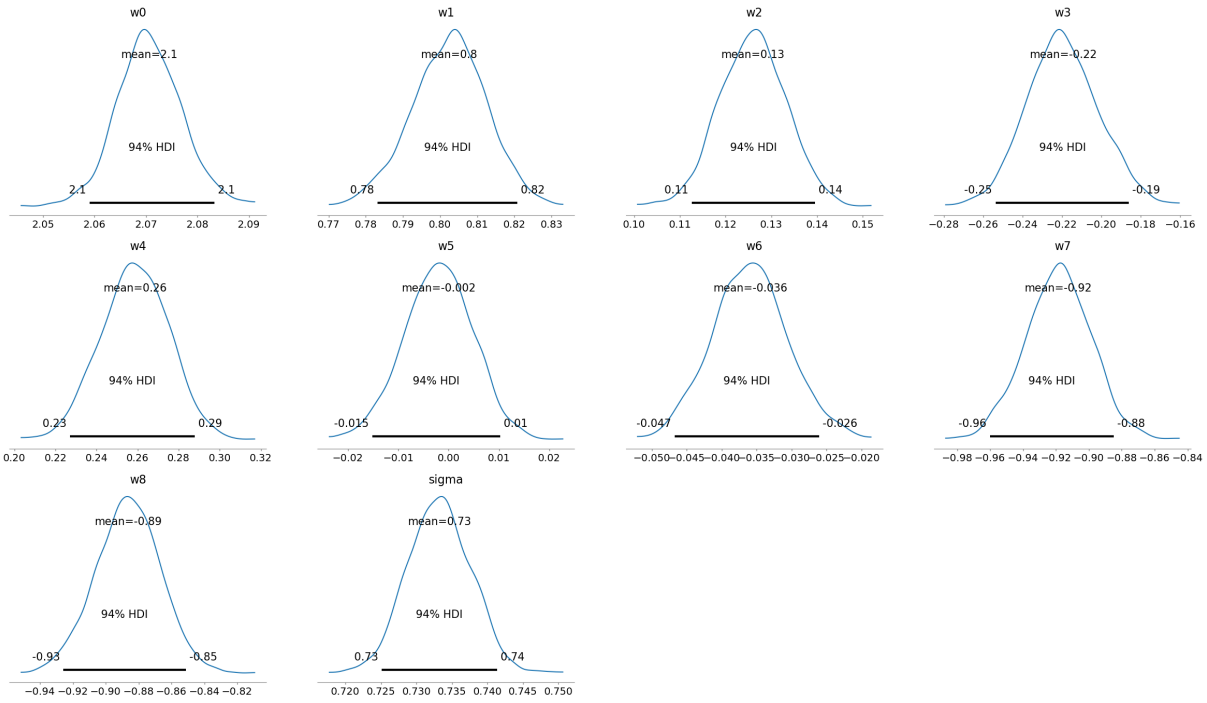

Figure 12

Using the mean parameters given by the posterior distributions, the predicted house values were calculated from the data-set, giving an R2 score of **60.1%**. As a comparison, the Least Squares Regression model from SKLearn was applied to the same data, yielding an R2 score which was almost identical, showing that in this instance, Least Squares Regression and Bayesian Linear Regression have a similar level of precision.

The model was run twice more, trained with random samples of 50 and 500 respectively with the resulting means and variances of the resulting posteriors shown in **Figures 13 and 14**. When making

predictions on the test data-set, the models yeilded R2 scores of **36.0%** and **58.6%** respectively

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
w0	2.109	0.118	1.885	2.312	0.003	0.002	1524.612	1392.909	1.001
w1	1.091	0.222	0.700	1.521	0.008	0.006	710.161	788.043	1.008
w2	0.079	0.124	-0.149	0.301	0.003	0.002	1332.111	1460.302	1.009
w3	-0.515	0.404	-1.206	0.287	0.015	0.012	745.223	747.251	1.009
w4	0.974	0.665	-0.270	2.175	0.021	0.015	964.838	1200.045	1.004
w5	-0.184	0.127	-0.424	0.048	0.003	0.002	1398.741	1171.125	1.001
w6	-1.962	1.864	-5.241	1.811	0.052	0.038	1264.249	1353.422	1.001
w7	-1.040	0.384	-1.716	-0.297	0.013	0.009	916.568	1085.150	1.003
w8	-0.944	0.393	-1.664	-0.179	0.013	0.009	891.989	1210.430	1.002
igma	0.711	0.080	0.568	0.855	0.002	0.002	1196.249	1460.965	1.003

Figure 13

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
w0	2.067	0.032	2.005	2.128	0.001	0.000	2255.141	1262.565	1.004
w1	0.840	0.056	0.731	0.944	0.001	0.001	1564.393	1177.694	0.999
w2	0.089	0.035	0.027	0.158	0.001	0.001	2064.508	1609.950	1.001
w3	-0.302	0.099	-0.485	-0.111	0.003	0.002	1536.562	1329.966	1.001
w4	0.574	0.115	0.364	0.788	0.003	0.002	1730.971	1438.890	1.001
w5	0.014	0.033	-0.046	0.075	0.001	0.001	2339.347	1533.542	1.005
w6	-1.709	0.294	-2.242	-1.152	0.006	0.004	2413.808	1265.727	1.002
w7	-1.009	0.099	-1.185	-0.812	0.002	0.002	1722.507	1728.293	1.003
w8	-0.998	0.096	-1.186	-0.829	0.002	0.002	1704.069	1570.788	1.003
sigma	0.690	0.021	0.651	0.730	0.000	0.000	2355.019	1389.200	1.002

Figure 14

These lower scores with less data show that the model requires more data to have more certainty in the posteriors. It can be see that the model trained on 50 samples has higher variances in its parameter distributions, displaying this lack of certainty.

4 Trees and Ensembles

4.1 CART Decision Trees

Decision Trees consist of decision nodes and leaf nodes in a tree structure. A given input is parsed by the tree being evaluated using the decision nodes until a leaf node is reached. Each decision node consists of one of the input variables in the data-set and a threshold which splits the data about this node; each leaf node represents an output class or target value depending on the type of output required.

The CART (Classification And Regression Tree) algorithm is a heuristic method which attempts to find the optimal structure for this tree which minimises the error of evaluating inputs. The root node is found by the split that minimises the mean squared error of the two branches. The weights are equal to the number of data points that satisfy the condition of the corresponding branch. The remaining decisions are found using the same method but only using data points which satisfy the conditions set by each ancestral node. The leaf nodes are created when either the mean squared error is 0, or some other criteria such as reaching a specified maximum depth.

I attempted to find the optimal hyper-parameters for the Decision Tree Regressor to be applied to the California Housing dataset. I used a Grid Search with cross validation to run through a number of different values for each hyper-parameter (keeping all others at their defaults). The Grid Search is more exhaustive than the Randomized search so likely gives slightly more optimized parameters with the tradeoff of increased fitting time. The box plot shown below in **Figure 15**indicates how the mean test score of the tree relates to different settings for each hyper-parameter. The dotted line indicates the mean test score with all parameters at their default settings.

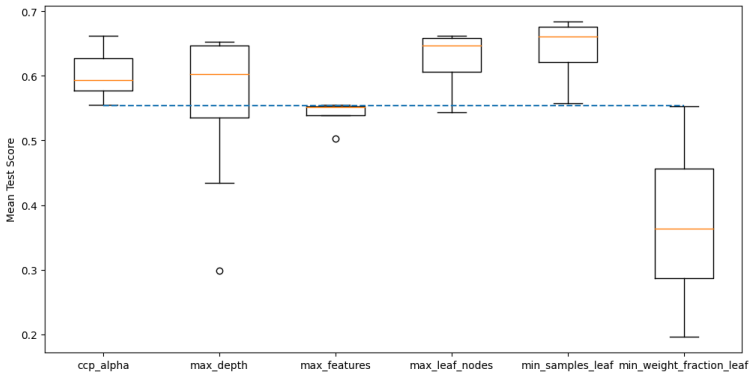

Figure 15

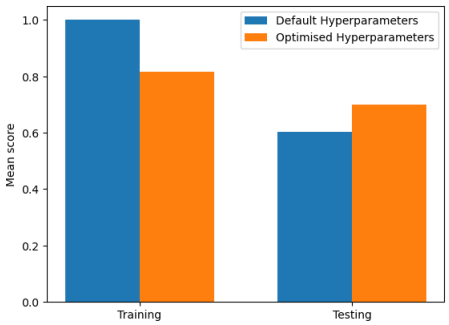

Figure 16

From this, it can be seen that the parameter with the strongest positive effect on the mean test score was the 'min_samples_leaf' parameter - the optimal value of which was shown to be 15. The 'max_depth' and 'min_weight_fraction_leaf' parameters had a higher variance of scores and therefore are stronger overall effect on the model, but the optimal value of the former was 'None' (the default value) and increasing the latter only resulted in a decrease in score.

The three hyper-parameters most capable of increasing the score were shown to be 'ccp alpha', 'max-leaf-nodes' and 'min samples per leaf', although applying found values of 'min samples per leaf' had a negative impact on score when used with the other two parameters. As such I focused on these other two and found values of **6.61e-05** for 'ccp alpha' and **15** for 'min samples per leaf'. The effect of using these two optimized parameters is shown in **Figure 16** against the model with default parameters, with the higher test score showing an improvement in performance and the lower training score showing a reduction in over-fitting.

The plots of both these parameters against training time is shown in **Figures 17 and 18**. Both of the chosen values which were selected based on score also appear to reduce training time.

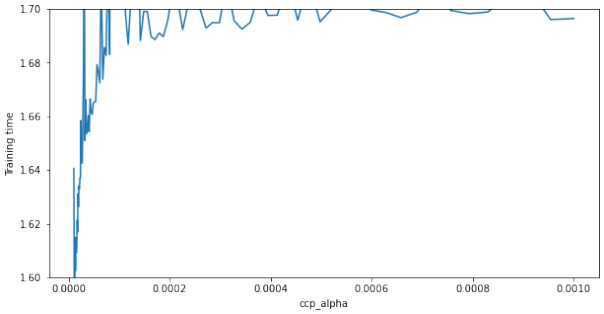

Figure 17

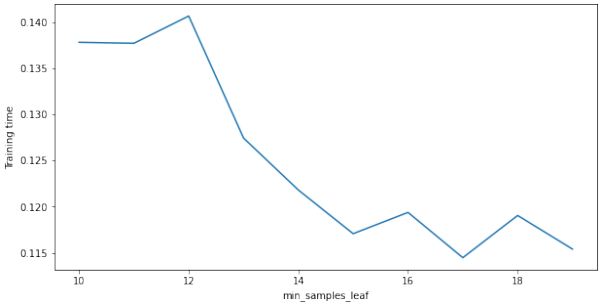

Figure 18

The model achieved a score of **71.8% on the test set** and this near 10% increase over that of the Bayesian Linear Regression shows that the Decision Tree is a more effective regression model in this scenario. Linear regression works well when the data follows a linear shape but as it cannot capture non-linear patterns, a decision tree is generally more powerful when used with a more complex data-set.

To gain an insight into how the decision tree predicts incorrect values, the predicted value with the greatest square distance from its true value was found. This point was observed as:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	-0.44888	1.856182	-0.268291	-0.048032	-0.88524	-0.118578	-0.717197	0.603801

with a mean squared error of **14.6**. The visualised decision tree in **Figure 19** shows that this point was mapped after three decisions, only concerning two features, to a leaf node.

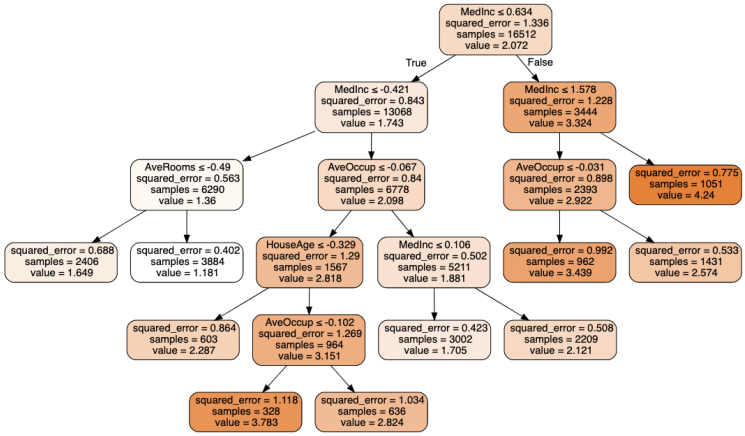

Figure 19

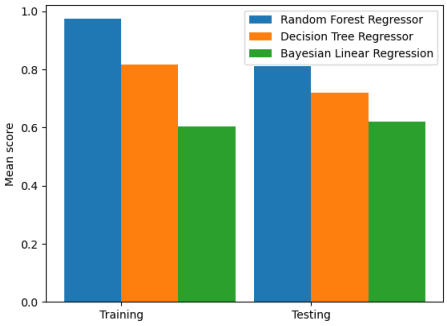

Figure 20

4.2 Ensemble Methods

Ensemble methods use a combination of models, each of which are trained on different samples, to make predictions. They can often be more powerful than single models as the expected error is usually less than that of the average model in the ensemble. This section, the aim is to use SciKit's Random Forest Regressor which trains a given number of decision trees on various samples and uses the average prediction of each tree as its overall prediction. Features towards the top of the tree contribute to the final prediction decision of a larger fraction of the input samples.

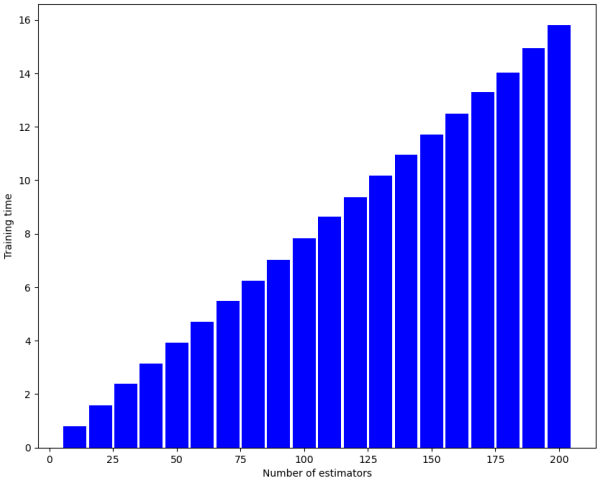

Figure 21

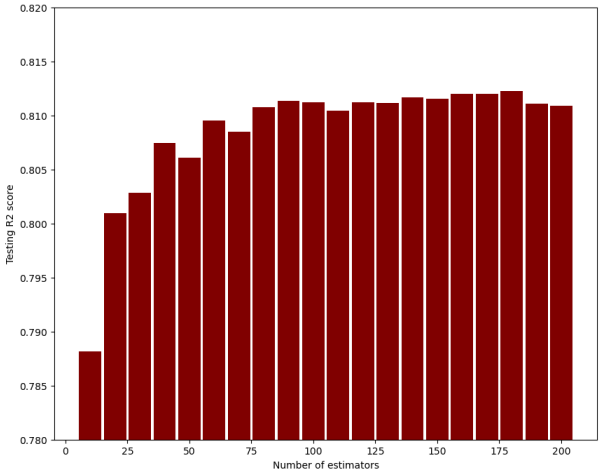

Figure 22

Figures 21 and 22 show the number of estimators (between 10 and 200) used effects training time and precision. The training time appears increase linearly with respect to this number but the mean test score stops improving significantly at around 100 estimators.

The prediction results of the previously used Bayesian Linear Regression and Decision Tree Regressor models were plotted alongside that of this Random Forest Regressor in **Figure 20**. The Forest Regressor is shown to perform best, likely because of bayesian linear regression does not provide a complex enough function to model the data as well as the tree and the forest regressor.