Report

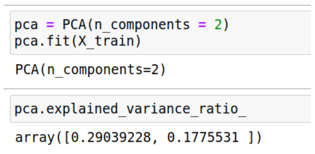
2.1. Fashion MNIST dataset

2.1.1. Principal Component Analysis

Principal Component Analysis (PCA) is a method commonly used for dimensionality reduction [1]. It is a process of changing the basis of the data in order to end up with dimensions with low variance which can be discarded, without losing too much information from the initial data. It can be thought of as a way of looking for projections with maximum variance onto a lower dimensional linear space. Dimensionality reduction is needed to cater for the curse of dimensionality, which refers to datasets that have high dimensions, which make interpretation more difficult, as well as not being able to produce good results, because very often the datasets do not cover the entire feature space.

From a mathematical point of view, the easiest example would be considering the case of reducing a D-dimensional space to a one-dimensional space (M=1) [1]. The direction of this space can be represented by a D-dimensional vector u_0 which has unit length, i.e. $u_0^T u_0 = 1$. Thus, the projection of a point x is $u_0^T x$, and the mean of the projected data becomes $u_0^T \bar{x}$, where $\bar{x}$ is the dataset's mean, $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$. The variance of the projected data can be thus calculated by using the vector u_0, therefore the variance becomes $\frac{1}{N}\sum_{i=1}^{N}(u_0^T x_i - u_0^T \bar{x})^2 = u_0^T S u_0$, where S is the covariance matrix $S = \frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})$. Maximizing the projected variance can be done by differentiating the variance with respect to u_0 and setting the derivative to 0, producing the equation $Su_0 = \lambda_0 \mathrm{u}_0$. This means that u_0 is an eigenvector of S, and that in order to maximize the variance u_0 needs to be the eigenvector with the biggest corresponding eigenvalue. The eigenvector then becomes the *first principal component*. Generalizing on the above method, the i-th principal component is the one which maximizes the variance subject to being orthogonal to the previous (i-1) components. Now, reducing the D-dimensional to an M-dimensional one becomes a job of finding the first M principal components that satisfy the above conditions.

The dataset comprises of images that have a shape of (28x28). This means that the number of dimensions is D = $28^2 = 784$ dimensions, which can make the computations costly, especially as the dataset is large, having 60,000 images that can be used for training. In order to observe the structure of the training set, a viable solution is to reduce the number of dimensions to a value of M=2, which now enables the plotting of the datapoints. The points will be coloured according to their class label. The result of applying PCA to the training set is the following:



```
pca = PCA(n_components = 2)
pca.fit(X_train)
```

```
PCA(n_components=2)
```

```
pca.explained_variance_ratio_
```

```
array([0.29039228, 0.1775531 ])
```

Figure 1 - Applying PCA and keeping the first two principal components

As can be observed from the Figure 1, the first two components account for 46.79% of the total variance.

Plotting the data projected onto the first two components produces the following result:

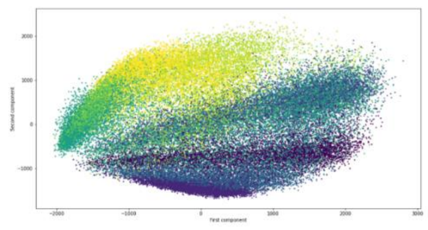


Figure 2 - The plot produced by the data projected onto the first two principal components

Analysing the plot, it can be deduced that there is high intra-class variance, as well as low inter-class variance. The class for which this is incredibly evident is the one coloured in yellow: it has a high intra-class variance, which can be deduced from the yellow points being incredibly spread out. It also has low-inter class variance, as it mostly overlaps with the lime coloured points, which means those two classes must have a lot of similarities.

This is an expected behaviour, as for instance an ankle boot can be incredibly similar to a sneaker shoe.

There are classes however, which can be clustered relatively easy, since they don't exhibit high intra-class variance (such as the purple coloured class at the bottom of the plot), and which exhibit much higher inter-class variance (the purple-coloured class at the bottom of the plot can be separated relatively easy from the light-blue coloured class adjacent to it). The deep-purple coloured class has a somewhat low intra-class variance, however it suffers from low inter-class variance, as there are a lot of points from other distributions that overlap with this one.

2.1.2. Gaussian Mixture Modelling

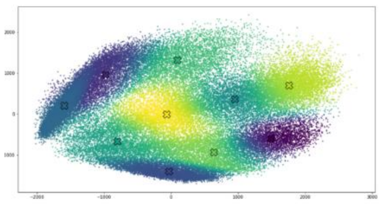


Figure 3 - GMM's soft clustering

Due to the shape and spread of the distributions, the best approach is to use soft clustering. Rather than assigning each datapoint to one of the 10 clusters, a list of probabilities for each cluster is assigned to each datapoint by using Maximum Likelihood Estimate (MLE) [2]. Thus, each point has a probability of belonging to a certain cluster, i.e. the probability that a certain Gaussian distribution has produced that point. This is all done by the Expectation Maximization (EM) algorithm. Each of the k distributions has a mean $u_k = \frac{1}{N_k}\sum_{n=1}^{N} g(z_{nk})x_n$. The probability of each cluster becomes $= \frac{N_k}{N}$. The covariance matrix is defined by $S_k = \frac{1}{N_k}\sum_{n=1}^{N} g(z_{nk})(x_n - u_k)(x_n - u_k)^T$. The function $g(z_{nk})$ is called the responsibility function and represents the probability of a datapoint x_n coming from the k-th distribution. Hence, in the E step of the EM algorithm, the responsibility function is calculated for each cluster using the current parameter values, while in the M step the parameters (i.e. the u_k, S_k, π) are recalculated using the newly computed responsibility function. These steps are repeated until convergence, with the E step being the first one in the algorithm. At the beginning of the algorithm, the parameters are randomly set.

Running soft clustering on the dataset produces the output shown in figure 3.

The colour for each datapoint has been decided using a function which does a linear combination of the weights of each cluster. Looking at the plot, it can be observed that it managed to produce some clusters correctly, while with others it has completely failed. For instance, the blue cluster at the bottom of the plot seems to have captured with a high precision the purple distribution that was in the original plot. This has been expected, as the original distribution didn't exhibit incredibly high intra-class variance, and had high inter-class variance. The points that were originally overlapping with this distribution are now coloured accordingly, which can be seen on the points that have a blue-green fade.

The deep purple distribution in the original dataset seem to be almost missing. This is due to the fact that it was mostly covered by points from other distributions.

The top left distributions seem to also resemble the original structure of the distributions; there are a lot of points which are coloured using a mix of the colours of the surrounding distributions, which infers that those points may be generally hard to classify, as they exhibit patterns which can place them in any of the nearby classes.

Overall, it seems like the soft clustering algorithm has done a fairly good job with the classes at the extremities, whilst the two central classes (represented by a dark blue and dark purple in the original plot) seem to have almost disappeared; this behaviour can be explained by the fact that those two distributions had the most overlapping points. The part where EM seems to have struggled the most is where in the original plot the yellow class overlapped the deep blue and dark green class which can be faintly observed: in the place of the overlap it seems like EM has thought that there was one of the distributions, having placed a yellow distribution there.

2.2.1.1. Artificial Neural Networks

Before creating the Multilayer Perceptron (MLP), there needs to be done some pre-processing. Firstly, PCA is applied to the dataset, but with the number of components being equal to the number of pixels in the image, i.e. 784. By plotting the number of features and how much variance they cover, it seems like a reasonable number of components would be around 40-60. Choosing M to be 40 seems to a good choice; as the first M=40 components amount for approximately 85% of the variance.

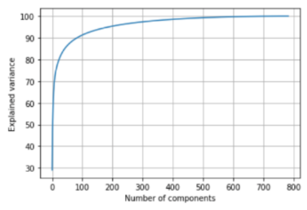


Figure 4-Plot showing the number of components and how much variance they express

Then, the datapoints are standardised by dividing them by the maximum value found in the training set (for the training points) and the testing set (for the points used for testing) in order to bring the values to a range between -1 and 1. The MLP classifier uses 3 hidden layers, each having the same number of neurons as the number of components in the projected dataset, meaning a total of 40 neurons per each layer. The activation function is the rectified function, as it produces faster learning compared to the sigmoid which has to do computationally expensive operations. The solver is 'adam', as this is the recommended one in sklearn's documentation for datasets which have thousands of training examples [3]. Learning rate is initialised to 1e-3, and so is the tolerance. The number of iterations where there is no change has been set to 25. Using the learning_curve [4] function provided by the model selection in sklearn enables to produce the following result, where the number of folds is 5, and there are 5 training set sizes, varying from 4800 to 48000.

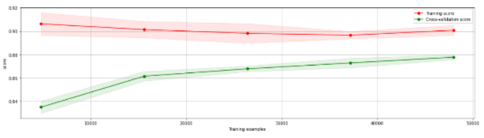


Figure 5- Train and Validation curves

The following results show that initially the training score is relatively high and the validation score is relatively low, indicating overfitting, but as the training size increases the training and validation scores converge to a value of roughly 90% for training, and 88% for validation, meaning there are no signs of overfitting in the end.

2.2.1.2. Testing accuracy

The testing score on the testing dataset is 86.54%. This further confirms that there's no overfitting with the model, as the difference between the training and testing accuracy is relatively small, and it performs well on unseen data.

2.2.1.3. Hyperparameters

In order to find the best hyperparameters for the model, sklearn's RandomizedSearchCV [5] is being used, which does hyperparameter tuning, i.e. selects the best parameters for a given list of parameters. Choosing the distribution for the alpha and learning_rate_init parameters and running 10 different fits, and plugging the produced values into a new model, produces a very similar testing score to the initial model.

Using sklearn's validation_curve [9] and varying just the learning rate parameter, the following pattern can be observed:

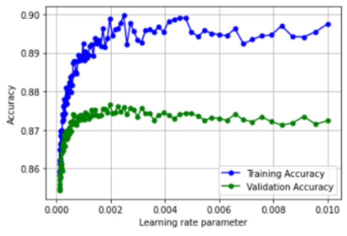


Figure 6- Validation and training accuracy with varying learning rate

As can be seen in Figure 6, low learning rates have a great impact on the accuracy of the model, however values greater than 0.002 do not make a drastic improvement in the validation accuracy, meaning that any value greater than that is a good fit for the model.

2.2.1.4. Decision boundaries

Decision boundaries help understand better how well the neural network is able to separate the classes.

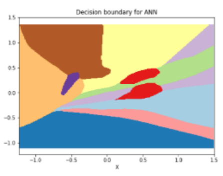


Figure 7- Decision boundary for the ANN

2.2.2.1. Support Vector Machines

Another way to tackle classification is through the use of support vector machines (SVM). They work by generating a hyperplane which maximizes the margins of the classes, through the use of kernels [6]. At a higher level, a kernel transforms non-linearly separable data into linearly separable data. Running the svm with the 'rbf' kernel and a regularization parameter C=10 produces the following output:

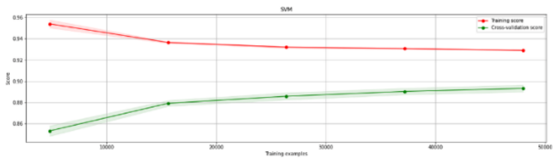


Figure 8- SVM train and test accuracy with rbf kernel and C=10

The observed behaviour is similar to the one of the artificial neural network: in the beginning the training accuracy is incredibly high and the validation accuracy is small, this means that initially, just like the MLP classifier, the model is overfitting, because of the small training set size. However, as the training size increases, the training accuracy converges to a value of 94%, while the validation accuracy converges to a value of approximately 89%, meaning that the model is able to generalize well on unseen data once the training size increases.

2.2.2.2. SVM testing accuracy

The SVM model scores 88.42% accuracy on the test data, which further supports the idea that the model is not overfitting, as it produces almost exactly the same result on unseen data as it did during validation.

2.2.2.3. Hyperparameter sensitivity

The kernel initially used is the Gaussian Radial Basis Function (rbf), as this works well with non-linearly separable data, for which there is no prior knowledge [7]. The figure below shows the results of applying different kernels with the same C value:

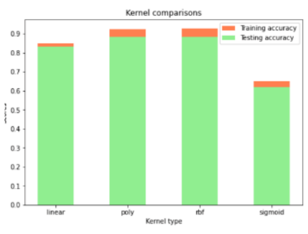


Figure 8- Training and testing accuracy with different kernels

Both the rbf and poly kernels seem to have done a pretty similar job in terms of accuracy. The sigmoid kernel has done the worst job, which could perhaps be explained by the fact that it works similar to an activation function in a neural network [7].

The C parameter is often chosen through trial and error. An incredibly low C value means it would ignore most of the class outliers, while a high C value would take into consideration most outliers when creating the class margin [8]. In figure 13, the effect of changing C can be observed on the rbf kernel:

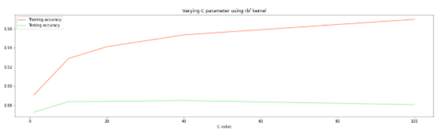


Figure 9 - SVM accuracy as C increases

Initially, with a small C value, the SVM has a relatively small testing accuracy. At C=10, the training and testing accuracy seem to have the smallest error, however as C increases to 100, the model overfits the data. This can be observed through the testing accuracy which converges to approximately 88%, while the training accuracy continues to increase, which happens because it caters for too many outliers when creating the decision boundary.

2.2.2.4 Decision boundary

The decision boundary helps understand how well the support vector classifier is able to separate the classes.

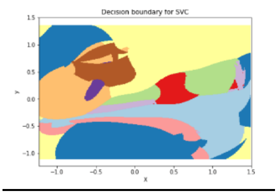


Figure 10 - Decision boundary for SVC

2.2.2.5 SVM and ANN comparison

To compare the SVM and artificial neural network, both the validation accuracy and their average training times have been taken into consideration. Figure 11 presents how they benchmark against one another.

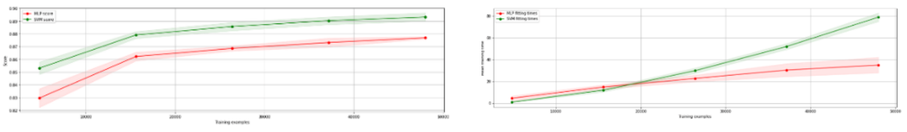


Figure 11- Left: Testing scores, Right: Fitting times

The SVM has higher accuracy, but at the same time considerably higher training time. Arguably, the artificial neural network is the better model for this dataset, as it takes much less time to train, whilst keeping the final accuracy 2% lower than the one of the svm, which in practice will not make a drastic difference. In the end, the difference in training times makes the neural network a solid choice, as long as the hyperparameters are also chosen carefully.

2.3 The California Housing dataset

2.3.1. Plotting the latitude and longitude with the median house value

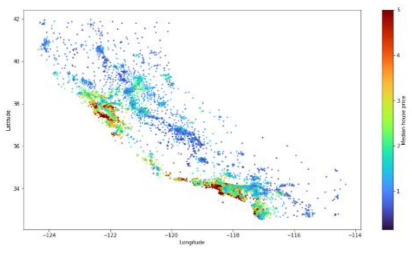
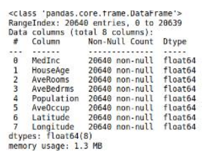


Figure 12- The latitude and longitude plot of the California dataset

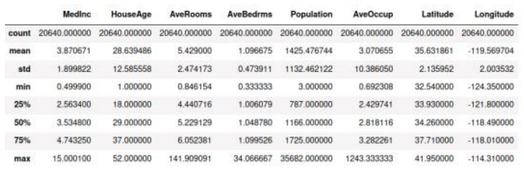
The latitude and longitude plot for the datapoints produces a pattern which is incredibly similar to the geographic shape of the California state. Analysing the plot, the areas that are have the most expensive houses coincide with (from top left to bottom right), Sacramento, San Francisco, Los Angeles and San Diego. The mentioned cities have the most population compared to the rest of the state, which gives further insight into the median house prices in that area.

2.3.2. Cleaning/Transforming the dataset

Panda’s dataframe offers a couple of useful functions: describe and info. Using info provides great insight into the structure of the dataset:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
dtypes: float64(8)
memory usage: 1.3 MB
```



	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
count	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000	20640.000000
mean	3.870671	28.639486	5.429000	1.096675	1425.476744	3.070655	35.631861	-119.569704
std	1.899822	12.585558	2.474173	0.473911	1132.462122	10.386050	2.135952	2.003532
min	0.499900	1.000000	0.846154	0.333333	3.000000	0.692308	32.540000	-124.350000
25%	2.563400	18.000000	4.440716	1.006079	787.000000	2.429741	33.930000	-121.800000
50%	3.534800	29.000000	5.229129	1.048780	1166.000000	2.818116	34.260000	-118.490000
75%	4.743250	37.000000	6.052381	1.099526	1725.000000	3.282261	37.710000	-118.010000
max	15.000100	52.000000	141.909091	34.066667	35682.000000	1243.333333	41.950000	-114.310000

Figure 13-Left: Information about the dataset, Right: Description of the dataset

The information provided by the function suggests that there is no need to clean the data. The non-null values in all columns are equal to the number of total datapoints, meaning that there aren’t any features that needed to be filled by taking their median value for instance. Likewise, the type for all features is float, therefore there is no need to do any processing, as it would have been the case with categorical features.
In terms of transforming the data, the describe function provides helps with that decision.
The difference between the means and deviations of all the features is quite extreme, and for instance a unit change in average number of bedrooms does not mean the same as a unit change in population. Therefore, data scaling is needed in order to cater for this issue, and sklearn provides a processing library [10] which brings all the features to a mean of 0 and a deviation of 1. Another step that needs to be taken is the shuffling of the dataset, as by printing the first 10 and last 10 datapoints a pattern can be observed in the latitude and longitude features, therefore shuffling is required in order to avoid biases.

2.3.3. Running Bayesian Linear Regression

The goal with Bayesian Linear Regression is to obtain the marginal posterior distributions over the parameters, which then would enable generating estimates for new datapoints. Before being able to get the posteriors, prior distributions need to be chosen. It was assumed that there was no prior knowledge of the true parameters, therefore the priors were chosen to have a mean of 0 and standard deviation of 40. The noise distribution has a deviation of 20 instead of 40. Running the No U Turn Sampler and then sampling with two chains produces the following posterior distributions over the parameters:

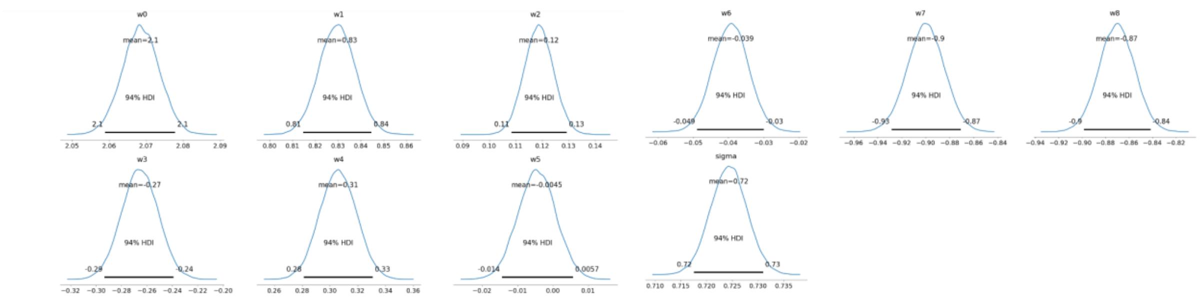
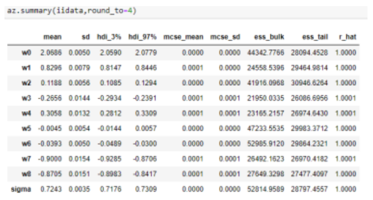


Figure 14- Results of running the NUTS sampler

Arviz provides a summary for the inference data, shown in the left image in figure 15.



```
az.summary(iidata,round_to=4)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
w0	2.0686	0.0050	2.0590	2.0779	0.0000	0.0000	44342.7766	28094.4528	1.0000
w1	0.8296	0.0079	0.8147	0.8446	0.0001	0.0000	24558.5396	29464.9814	1.0000
w2	0.1188	0.0056	0.1085	0.1294	0.0000	0.0000	41916.0968	30946.6264	1.0000
w3	-0.2656	0.0144	-0.2934	-0.2391	0.0001	0.0001	21950.0335	26086.6956	1.0001
w4	0.3058	0.0132	0.2812	0.3309	0.0001	0.0001	23165.2157	26974.6430	1.0001
w5	-0.0045	0.0054	-0.0144	0.0057	0.0000	0.0000	47233.5535	29983.3712	1.0000
w6	-0.0393	0.0050	-0.0489	-0.0300	0.0000	0.0000	52985.9120	29864.2321	1.0000
w7	-0.9000	0.0154	-0.9285	-0.8706	0.0001	0.0001	26492.1623	26970.4182	1.0001
w8	-0.8705	0.0151	-0.8983	-0.8417	0.0001	0.0001	27649.3298	27477.4097	1.0000
sigma	0.7243	0.0035	0.7176	0.7309	0.0000	0.0000	52814.9589	28797.4557	1.0000

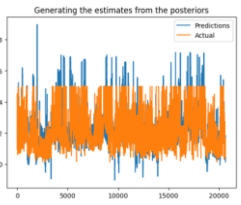
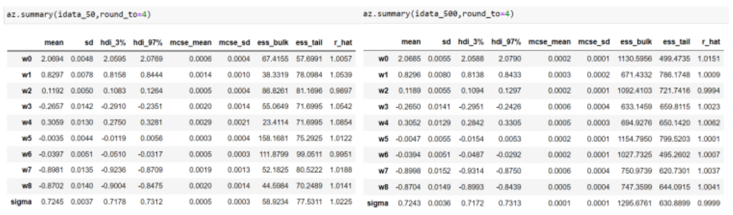


Figure 15- Left: Means and deviations for the distributions, Right: Plotting the predictions against the real house values

2.3.4 Results of the Bayesian Linear Regression

Using the means from the learned posterior distributions to generate the predictions, and plotting that against the real values produces the results shown in the right image in figure 15. The plot confirms that pymc has produced a good approximation of the desired distributions, given that using the means of the posterior distributions captures quite well the shape of the actual house values.

2.3.5. Using 50 and 500 samples with pymc



```
az.summary(idata_50,round_to=4)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
w0	2.0694	0.0048	2.0595	2.0769	0.0006	0.0004	67.4155	57.6991	1.0057
w1	0.8297	0.0078	0.8158	0.8444	0.0014	0.0010	38.3319	78.0984	1.0539
w2	0.1192	0.0050	0.1083	0.1264	0.0005	0.0004	86.8261	81.1696	0.9897
w3	-0.2657	0.0142	-0.2910	-0.2351	0.0020	0.0014	55.0649	71.6995	1.0642
w4	0.3059	0.0130	0.2750	0.3281	0.0029	0.0021	23.4114	71.6995	1.0854
w5	-0.0035	0.0044	-0.0119	0.0056	0.0003	0.0004	158.1681	75.2925	1.0122
w6	-0.0397	0.0051	-0.0510	-0.0317	0.0005	0.0003	111.8799	99.0511	0.9951
w7	-0.8981	0.0135	-0.9236	-0.8709	0.0019	0.0013	52.1825	80.5222	1.0188
w8	-0.8702	0.0140	-0.9004	-0.8475	0.0020	0.0014	44.5984	70.2489	1.0141
sigma	0.7245	0.0037	0.7178	0.7312	0.0005	0.0003	58.9234	77.5311	1.0225

```
az.summary(idata_500,round_to=4)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
w0	2.0685	0.0055	2.0588	2.0790	0.0002	0.0001	1130.5956	499.4735	1.0151
w1	0.8296	0.0080	0.8138	0.8433	0.0003	0.0002	671.4332	786.1748	1.0009
w2	0.1189	0.0055	0.1094	0.1297	0.0002	0.0001	1092.4103	721.7416	0.9994
w3	-0.2650	0.0141	-0.2951	-0.2426	0.0006	0.0004	633.1459	659.8115	1.0023
w4	0.3052	0.0129	0.2842	0.3305	0.0005	0.0003	694.9276	650.1420	1.0062
w5	-0.0047	0.0055	-0.0154	0.0053	0.0002	0.0001	1154.7950	799.5203	1.0001
w6	-0.0394	0.0051	-0.0487	-0.0292	0.0002	0.0001	1027.7325	495.2602	1.0007
w7	-0.8998	0.0152	-0.9314	-0.8750	0.0006	0.0004	750.9739	620.7301	1.0037
w8	-0.8704	0.0149	-0.8993	-0.8439	0.0005	0.0004	747.3599	644.0915	1.0041
sigma	0.7243	0.0036	0.7172	0.7313	0.0001	0.0001	1295.6761	630.8899	0.9999

Figure 16- Running pymc with 50 and 500 samples

The main observable differences are in the mean and standard deviation of the distributions. The model which ran using just 50 samples seems to be less close to the actual values of the distributions compared to the model which ran using 500 samples. This is expected, since it needs much more samples to learn the posterior distributions.

2.4.1. CART Decision Trees

2.4.1.1. CART Trees explained

CART Decision Trees are machine learning models which handle both classification and regression tasks. It works by creating a tree where each node that is not a leaf node represents a decision on the features of the datapoint and the leaf nodes represent either class labels in the classification case, or they represent scalar values in the context of regression.

2.4.1.2. Model selection

To perform model selection, hyperparameter tuning was used on the criterions, maximum number of leaves and the maximum depth of the tree. Using RandomizedSearchCV to find the best parameters, the best result is given by the Poisson criterion, with a maximum depth of 20 and 180 maximum leaf nodes.

In terms of the strongest effect on the model, the maximum depth hyperparameter has the greatest impact on the base model (i.e. the model with the default options).

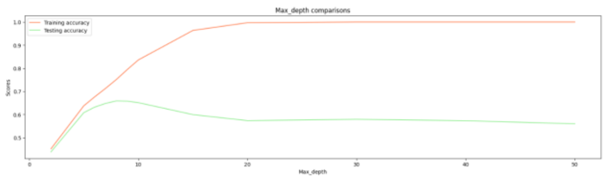


Figure 17 - Model accuracies with varying depth

The best testing accuracy on the base model seems to be given by a maximum depth of around 9, whereas with values greater than that it can be observed that the testing accuracy drops to around 64%, and the training accuracy gets to 100%, which is a clear sign of an overfitting model.

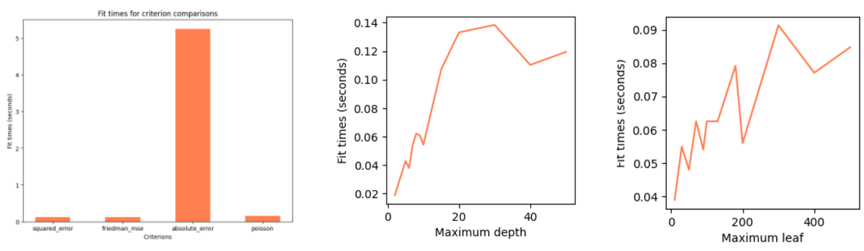


Figure 18- Left: Criterion times, Middle: Depth times, Right: Leaf times

2.4.1.3 Training times based on the hyperparameters

In terms of fitting times, choosing a value for maximum depth or the maximum number of leaf nodes does not impact the performance in terms of time taken to fit the model, however the choice of criterion has a significant impact, with the absolute error dominating the time taken to fit, with a value of approximately 5 seconds. Given the size of the dataset and the training set implicitly (80% of total dataset), Poisson is the best choice, as it provided the best result accuracy wise, and hyperparameter tuning also recommended this criterion, and time taken to fit the model is under a second.

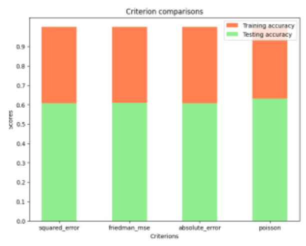


Figure 19 - Accuracies for different criterions

2.4.1.4. Scoring the Decision Tree

The accuracy of the setup given by the tuning is 71.9% on the test, and a training score of 79.61%. The results suggest that there is a small amount of overfitting, but based on the training score it also implies that the model is not capable of finding good node splits, since the training score is rather low.

2.4.1.5. Advantages of Decision Trees

Compared to the linear regression used previously, a Decision Tree is less computationally expensive, as it uses less memory than a Markov chain. They also don't require feature scaling, and with complex datasets they might be easier to interpret and understand the results, by following the decision nodes to the leaf.

2.4.1.6. Following path for misclassified point

The second point in the training set is estimated to have a house value of 3.970, when its actual value is 3.585. Following from the root node which is a decision on the MedInc feature, the datapoint follows the path up on the right of the root up to MedInc <= 8.786 (actual value is 8.3014). It goes left and reaches the decision on the house age, and given its house age value of 21, it goes on the left path from house age <= 27.5. Finally, the leaf node is reached, with the estimate of 3.970. The location of the house is in one of the denser areas with plenty of expensive house, which explains the reason it was priced highly. Perhaps an extra decision on the population following the house age could better predict the house value for that particular location.

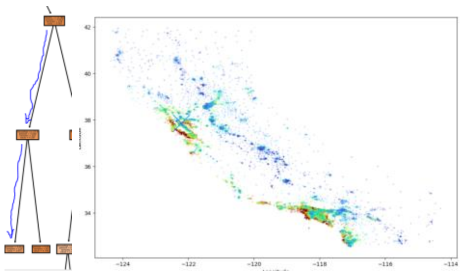


Figure 20- Final decision nodes starting from MedInc <= 8.786, and location of datapoint in terms of geographical coordinates, marked with a blue X

2.4.2. Ensembles

2.4.2.1. Random Forests

The chosen ensemble is represented by random forests, where each decision tree has the same hyperparameters as the initial decision tree, except each model in the ensemble uses a maximum number of features, instead of using all of them. The key benefit of this approach is that the errors of each model are not correlated to one another, because they don't use the same features when creating the node splits. This way, a more robust model is obtained. Sklearn provides the RandomForestRegressor model [11], where the hyperparameters from the original Decision Tree were kept.

2.4.2.2. Performance based on number of models

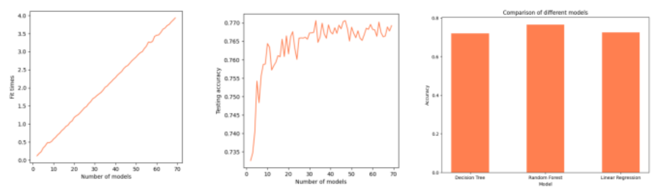


Figure 21- Left: Fit times, Middle: Ensemble performance, Right: Comparison for decision tree, random forest and linear regression

Based on the left and middle plots in Figure 21, the best number of models in the ensemble is 30, as it reacheas the peak accuracy, and any number of models greater than that oscilate around that value, but they don't surpass it. Given the left plot in Figure 21, it can be observed that the fitting time keeps increases with the number of models, but an ensemble with 30 models seems to keep the fitting time at less than a second, making this number of models the optimal one for this dataset.

2.4.2.3. Comparison the models

The ensemble has scored 76.65% on the test data, which is much more than the single Decision Tree has managed to score. From the right plot in Figure 12, the Random Forest is the model that has had the best performance on the California Dataset, therefore the Random Forest Regressor seems to be the optimal model for this particular task. It is also less computationally expensive compared to the Linear Regressor, and it takes under a second to fit 30 models. However, as the No Free Lunch Theorem states, there is no single best model for a predictive problem such as regression, and therefore the model chosen also comes down to the prefered method of analyzing the data and the resources available.

References:

1. Bishop, C.M., Pattern recognition and machine learning(2006), p. 561-563
2. https://github.com/uob-COMS30035/uob-COMS30035.github.io/blob/main/JamesLectures/gaussem.pdf
3. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html
4. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html
5. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
6. https://monkeylearn.com/blog/introduction-to-support-vector-machines-svm/?fbclid=IwAR1Tcnf-tYFiW8jznDfLwed9aOKf-qnwKZvJlGCRw7rOLj12jutGgpYYra8
7. https://dataaspirant.com/svm-kernels/?fbclid=IwAR0V0K06o7uK9ro7vj00njpIQqshdN6FLP5sQ221ihvHXC9u12GaELQlLSI
8. https://www.quora.com/How-can-I-choose-the-parameter-C-for-SVM?fbclid=IwAR2S-5IfxbdyzPix-8hoemMwUG-ReVLgi1RetfDZj04A1oS8p7_aiKVQEEc
9. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.validation_curve.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html