Machine Learning Coursework 2023

George Atkinson (rp21039)

Introduction

Machine Learning is a broad discipline which unequivocally will define the future of the world. This report aims to explore and analyze several machine learning classification models.

Upon the conclusion of this report, I hope to have learnt about and experimented with many machine learning techniques and have evaluated them for their strengths and weaknesses.

Section 1 – Classification and Sequence Labelling

In this section, I will use a dataset that tracks the activity of older people whilst they wear a sensor. This sensor records information for the frontal acceleration, vertical acceleration, lateral acceleration, RSSI (Received Signal Strength Indicator), phase, and frequency. For each data point, there is an associated action that the wearer is doing – this can either "sit on bed," "sit on chair," "lying," or "ambulating." My goal now is to use the data to make accurate predictions on what the wearer was doing.

Task 1 – Neural Network Classifiers

To begin with I set out to use sklearn's MLPClassifier (Neural Network Classifier) to classify my data. First, I split up the dataset into training data and testing data. Then, on each epoch I created a new validation set to enable me to plot a validation loss curve. A Neural Network can be configured in many ways, so I began with default hyperparameters and later changed variables to study its impact.

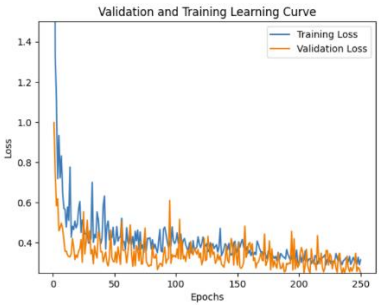


Figure 1: Validation and training learning curve with default hyperparameters

Once I ran the MLPClassifier with the default values, I plot a training and validation learning curve. These curves calculate the log loss of our classifier's predictions on each epoch, and we can use them to determine how well the model is performing. The log loss is a way of measuring the distance between the predicted probability of a classification and the ground truth classification. This makes the learning curve useful to see how accurately the model is correctly predicting classifications.

The training curve shows the model's performance on the training dataset, whilst the validation curve shows the performance on the validation set. The validation curve is a better measure of how robust the model is because it has to learn from new, unseen data.

In general, a learning curve (when plotting log loss) should show a generally negative trend because as the model iterates through the data more, it should learn more and achieve better predictions resulting in less loss. Despite a bit of noise, our model follows this trend as its log loss fluctuates around a value of 0.35. In addition, since we can see both curves trending downwards and staying close together, it means that the model is generalizing to new data well.

To analyze my results on the test set I firstly used sklearn's accuracy_score to check what percentage of predictions are correct. By doing this, I found that my MLPClassifier with default parameters achieved an accuracy of 89.4%.

As well as this, I used sklearn to plot a confusion matrix which can be seen in figure 2. The figure also contains a scaled confusion matrix which can be used to see patterns more obviously in the data.

By inspecting the confusion matrices, we can see a clear peak in predicted labels on the leading diagonal of the matrix which means that the model classified most of the datapoints correctly. However, by looking at the unscaled confusion matrix, we can see that the model is significantly worse at predicting datapoints in state 3. This is most likely due to a lack of training datapoints for state 3 causing the model to be bias towards the other states.

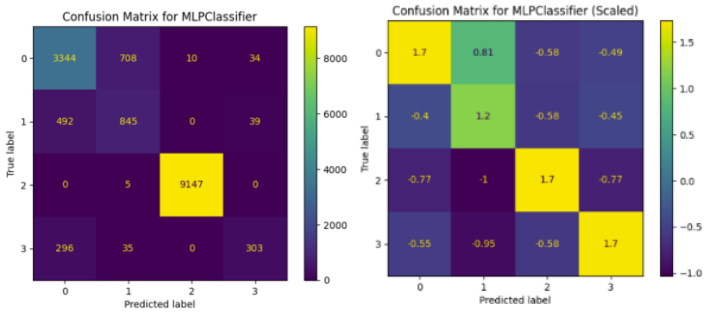


Figure 2: Confusion matrices, one scaled(right) and one unscaled(left) showing true label vs predicted label.

With the MLPClassifier there are lots of hyperparameters that have a large effect on performance. In my experiments I will experiment with changing the values of hidden layer sizes (HLS), batch size, and activation function. To begin with I altered the HLS and plotted a training loss curve which can be seen in figure 3. I plotted each line on the same graph so we can see which performed best. From the graph we can see that despite starting on the highest loss, the hidden layer size of (20,20) quicky produced the best loss score. It appears that for this dataset, smaller HLS produced

better results. This most likely means that there aren't very complex patterns in this dataset.

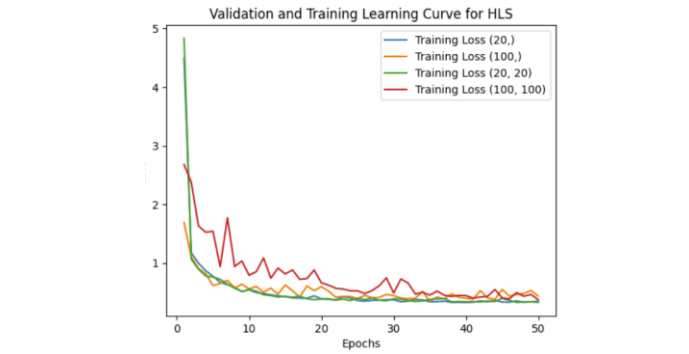


Figure 3: Validation and training curve comparison when changing HLS

Next, I changed the batch size and compared the training loss curves like in the last figure. With batch size I found that smaller batch sizes produce the least training loss and faster convergence, however, after around 50 epochs they all converge to a similar loss value.

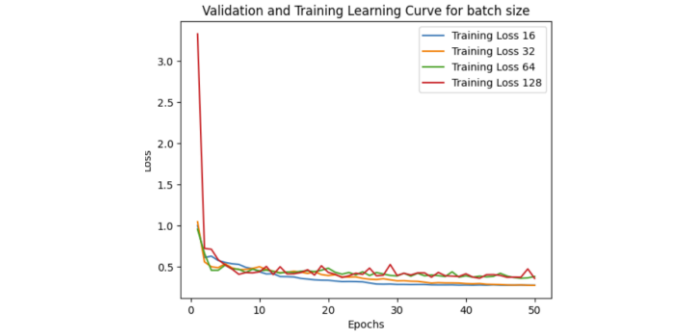


Figure 4: Validation and training curve comparison when changing batch size

Finally, I altered the activation functions to analyze the effect on performance. From this, I found that tanh and logistic had the least training loss whereas relu was more noisy.

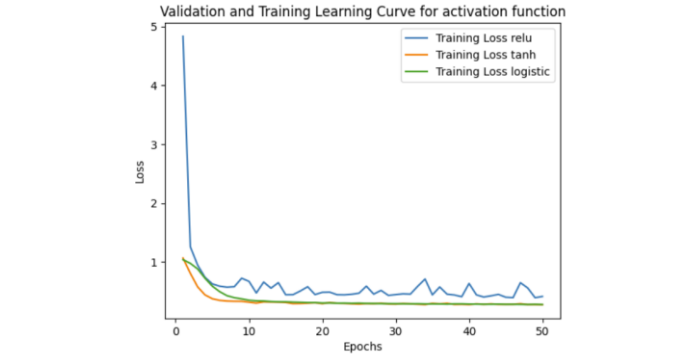


Figure 5: Validation and training curve comparison when changing activation function

Task 2 – Ensemble and Decision Trees

Another method of classification is by using decision tree classifiers. In this section I will analyze the performance of decision tree classifiers using different hyperparameters and compare their results with bagging ensembles of decision tree classifiers.

Decision trees are a popular choice for classification because they are quick, simple to understand, and computationally efficient. They work by recursively splitting the data into smaller subsets based on some features. This can be seen in figure 6 with a small decision tree which breaks up the dataset into 4 different classifications.

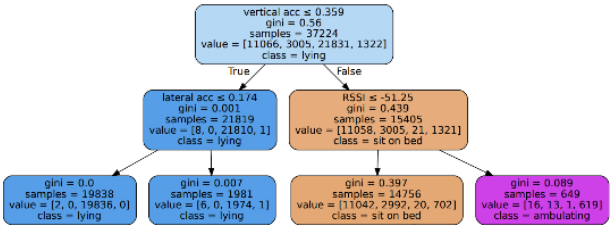


Figure 6: Decision Tree Max Depth = 2

Despite their haste, they are also prone to overfitting because they pick up on small fluctuations and become bias to some features and datapoints. This makes the model quite weak at making predictions on new data and can worsen the performance of the decision tree.

Bagging is a method in which you train multiple decision trees on different training data, and then combine the results of each tree, before finally taking the most popularly voted outcome. Whilst it may be more computationally expensive, it often does a great job at cancelling out any biases, leading to a more robust model.
The improved performance of the ensemble can be seen in figure 7. In the table in figure 7 it is clear to see that increasing the number of trees in increases the accuracy significantly from the accuracy for a single decision tree. This is because the biases of the individual trees will cancel out, which will enable the model to generalize better to new data.

Trees used	Accuracy
1	0.89
5	0.91
50	0.93

Figure 7: A table showing the number of trees used in ensemble improving accuracy.

Furthermore, figure 8 shows a more detailed relationship between model accuracy (when tested on the test set) and the number of decision trees in the bagging ensemble. In this case, our singular decision tree achieved accuracy of 0.897, whereas our ensemble achieved a peak accuracy of 0.932 and a mean accuracy of 0.922. From this, it is clear to see that bagging has improved the performance of our model on the test set.

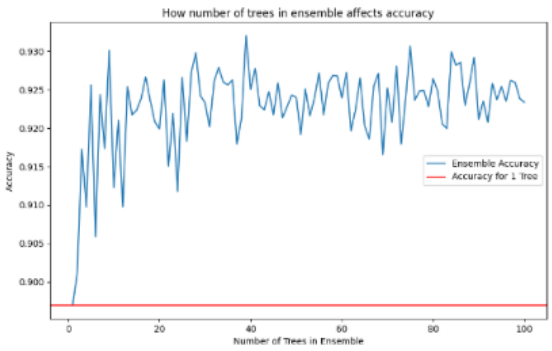


Figure 8: a graph showing the relationship between size of an ensemble and accuracy – plotted against accuracy of a single tree.

However, there are other considerations – for example, our bagging model for 40 models (the peak performance) took 0.172 seconds whilst our individual model took 0.018 seconds to run. Furthermore, although bagging certainly increases accuracy – this increase quickly turns into a noisy plateau for any ensemble with more than 25 models.
As previously mentioned, decision trees are so prone to overfitting because they become bias towards specific datapoints or features. Bagging is a great way to reduce the error in decision trees because by letting them all vote towards an outcome their biases tend to cancel each other out.

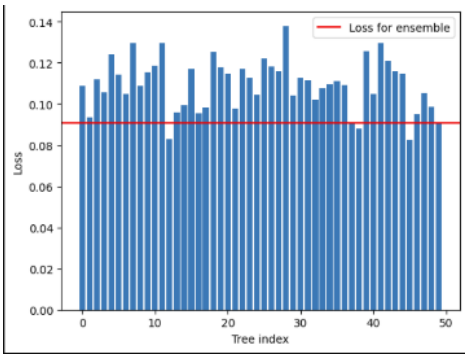


Figure 9: a graph showing the loss of individual trees plotted against the loss of the ensemble.

This effect is exhibited in figure 9 which plots the individual loss of 50 trees that were included in a bagging ensemble. Also on the graph is a line (red) to show the loss of the ensemble as a collective. From this graph we can see that the ensemble as a collective outperforms 47 out of the 50 individual trees.
Moreover, it manages to reduce the average trees loss of about 0.11 to a group loss of just 0.09. This just goes to show that bagging ensembles are successful in evening out any bias in the individual trees.

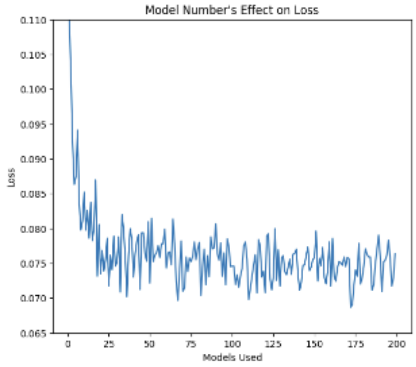


Figure 10: a graph showing the effect of the number of models in ensemble on loss.

Also, I found that the increasing the number of decision tree models in an ensemble improves performance but only to a certain extent. This can be seen in figure 10 which shows a negative correlation between number of models used and loss. However, for any number of models used beyond about 25, the loss just fluctuates between 0.07 and 0.08. For this reason, I can conclude that bagging improves the performance of the model over a single tree, but this improvement is limited.
The performance of our decision trees can be significantly affected by the hyperparameters that we initialize it with. There are four main hyperparameters that can be set: max_depth, min_samples_split, min_samples_leaf, and the criterion for measuring the quality of a split.

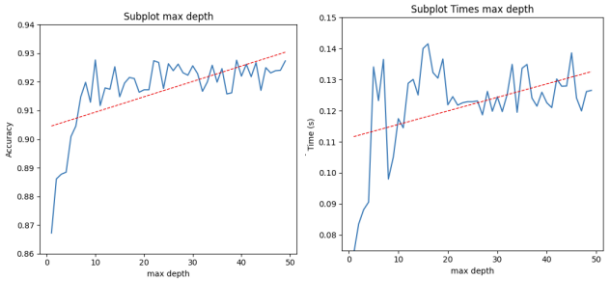


Figure 11: graphs showing max depths effect on accuracy (left) and runtime (right)

To begin with, we have the max depth of our tree. This determines the maximum number of layers the tree can have. From figure 11 we can see that by increasing the maximum depth of our tree we are able to increase the accuracy of the model. This is because by capping the max depth our model was underfitting its results leading to lower accuracy.
Although our model seems to plateau in performance quite quickly, max depth can be useful for improving accuracy on other models when the model is overfitting. It is also worth mentioning that whilst our program runs faster for exceptionally low max depths, the difference after max depth of about 5 is negligible.
Min samples split determines how many samples must be in a node for a split to be considered. By increasing this

parameter, we can reduce overfitting because it prevents splitting samples which don't have enough data to make a reliable decision.

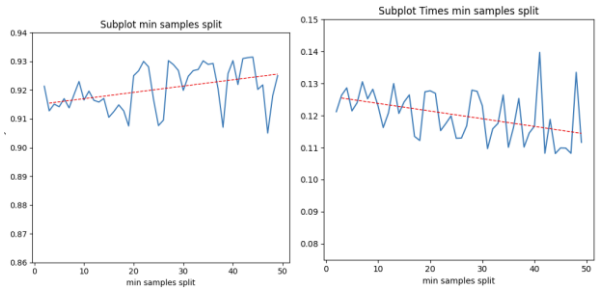


Figure 12: graphs showing the min samples split effect on accuracy (left) and runtime (right).

From our results in figure 12, we see a practically negligible change in accuracy when changing the min samples split. However, the time taken to run the program decreased by about 12%. Although a small improvement, when making larger trees, this could significantly improve performance time.

Min samples leaf is the parameter that specifies the minimum number of samples a node must have. By increasing this parameter, you can help to reduce overfitting because it prevents creation of leaf nodes with very little data to make decisions.

But, in our results (in figure 13) we saw a decrease in accuracy when increasing the min samples leaf. This probably means that our model wasn't overfitting and so it instead just hindered the amount of beneficial fitting it could do. Although, similarly to min samples split, by increasing the min samples leaf we were able to reduce runtime by a significant 25%.

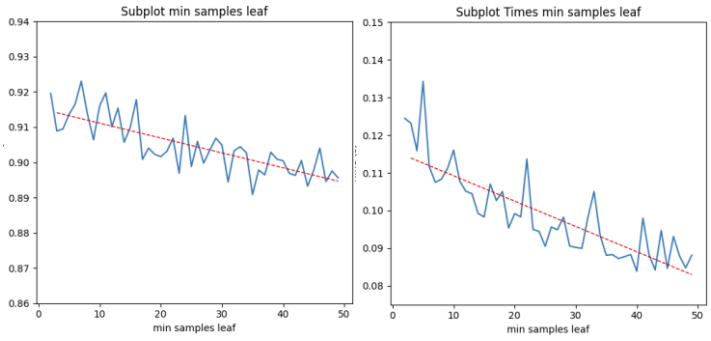


Figure 13: graphs showing min samples leaf effect on accuracy (left) and runtime (right)

By looking at all the graphs side by side (in figure 14), we can clearly see that max depth is the most sensitive hyperparameter because it causes the largest change in both accuracy and runtime. After max depth, min samples leaf has the next most significant effect – showing an improvement in training time although a slight decrease in accuracy.

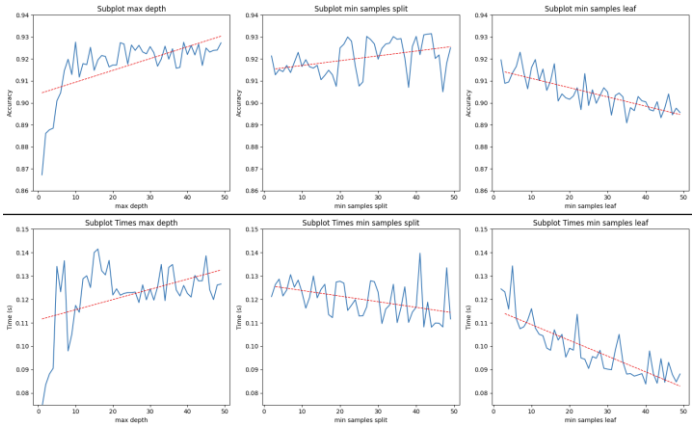


Figure 14: figures 8, 9, and 10, on the same scale for comparison.

Task 3 – Hidden Markov Models

Hidden Markov Models are a common statistical model that uses unsupervised learning for the task of labelling sequences of data in which the true state of the sequence is unknown ('hidden,' if you will).

HMMs have a set of hidden states as well as two important sets of probabilities associated with them: transition probabilities and emission probabilities. The transition probabilities show the likelihood of moving from one hidden state to another, and emission probabilities define the likelihood of an observation from the model. The HMM uses the Forward-Backward algorithm to predict the emission probabilities.

I initialized a GaussianHMM from hmmlearn, and then trained it on the X training dataset, providing it with a means matrix and covariance matrix from the dataset. Then by using the model to predict the states of the X test set, it produced an accuracy score of 93% and was very quick to train.

To further analyze my model's performance on the test set I plotted two confusion matrices (see figure 15). In the scaled confusion matrix, it is clear to see that the model is very accurate at predicting the labels because of the significant yellow colors on the leading diagonal.

However, it is also useful to look at the unscaled confusion matrix to truly see how few labels the model gets wrong. Also, by looking at the unscaled confusion matrix, we can see that the model is significantly worse at predicting datapoints in state 3 – perhaps due to the lack of training data available.

The transition matrix in figure 17 is a helpful representation to understand the chance of one state changing to another state. By cross-referencing the transition matrix with the emission graphs, we can further our understanding of the relationships between the states.

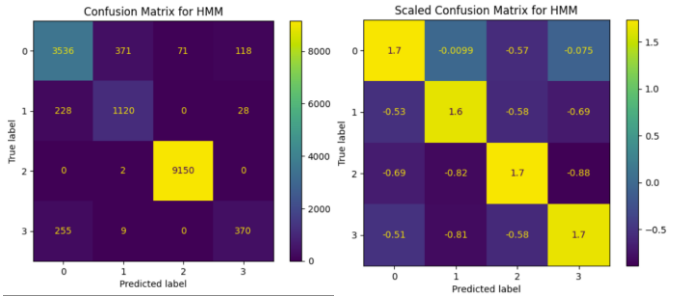


Figure 15: Confusion matrices, unscaled (left) and scaled (right), to show the true labels vs the predicted labels

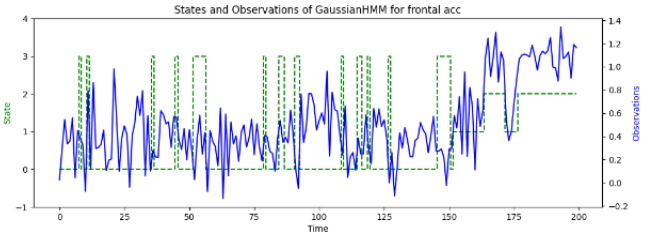


Figure 16: Emission graph for HMM on "frontal acceleration" feature (feature 0)

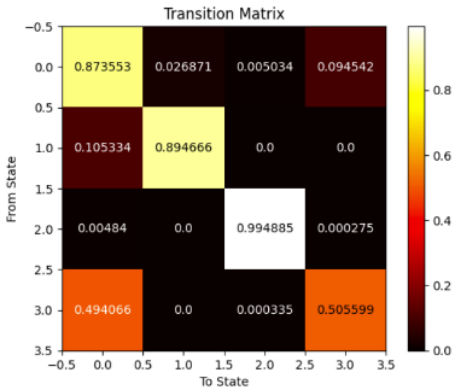


Figure 17: Transition matrix for HMM

By looking at the leading diagonal, we can see the probability of the state remaining the same is very high which explains why the state line in the figure is not jumping around constantly. This can especially be seen with state 2 whose probability of remaining the same is remarkably high – and when referenced with the previous figure we see that the HMM tends to stay in state 2 for a long time.

Furthermore, we can see the opposite effect with state 3 which has approximately 50% chance of staying the same and 50% chance of going to state 0 which is why we see this transition so frequently.

By recalling that the states are to do with activities in older people, it makes sense that the states aren't changing too often. Perhaps if the tracker was linked with sport, it would fluctuate between states a lot more. However, in the context of the older people with activities such as lying down and sitting on bed, these aren't going to be changing too frequently.

Figure 16 and 18 show the emission graphs on a sample of 200 for our frontal acceleration and vertical acceleration, respectively. From the graphs, we can see that frontal acceleration is significantly higher in state 2 whilst vertical acceleration is lower. Although I could make some deductions, these graphs ultimately aren't enough to make significant inferences on the states with more subtle changes.

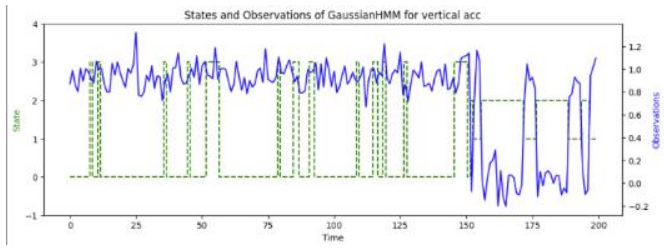


Figure 18: Emission graphs for HMM on "vertical acceleration" feature (feature 1)

To identify particularly prominent features for our model, I decided a means matrix would produce a far clearer overview of our features. I took the means matrix generated by hmm_model, which produced the image in figure 19. In this image each square shows the mean of a feature (on x-axis) when in the predicted state (on y-axis).

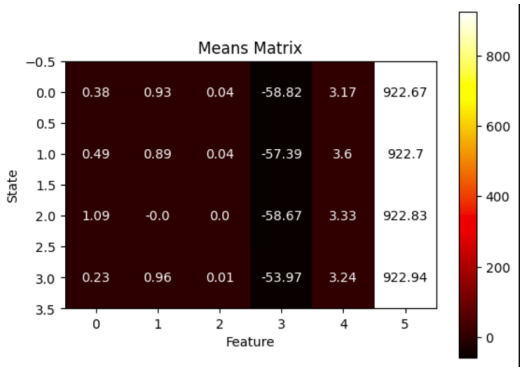


Figure 19: Means matrix for the states per feature.

Whilst this data was informative, any patterns weren't immediately obvious. For that reason, I then scaled the data and displayed it as an image. This can be seen in figure 20.

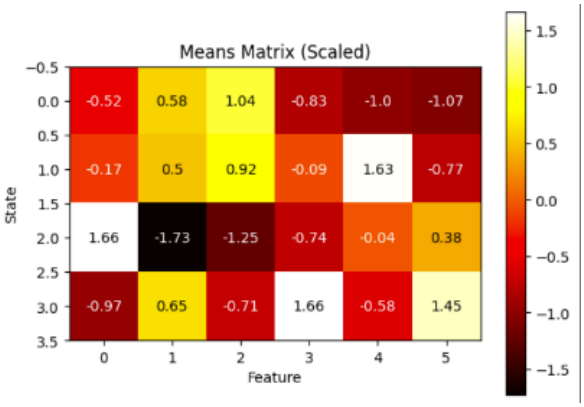
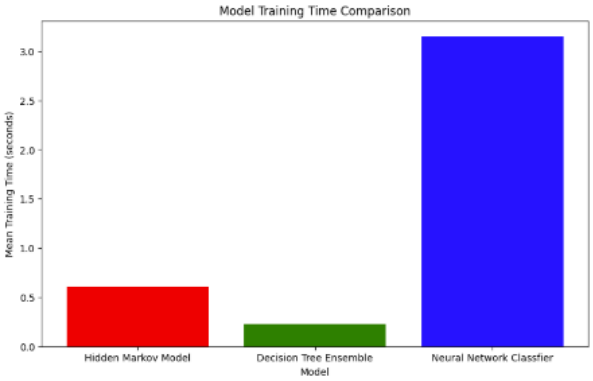


Figure 20 – Scaled means matrix for the states per feature

By inspecting the image, we can easily identify outliers. For example, by looking at feature 1, we can clearly see that this feature is significantly lower when in state 2. From this we

can infer that feature 1 is influential for deciding state 2. Moreover, we can even see patterns in the less clear states. As an example, state 0 has a fairly standard value for every feature. In the previous emission graphs, this would be unnoticeable, but by looking across every feature our matrix, we can more easily deduce what the features should look like while in state 0.

Upon gathering similar data about these models, it is natural to wonder which one is the best. With this, the three key areas we should compare for each model are the performance, the training time, and the interpretability. I have collated data on training time and accuracy in figure 21 which will be referenced throughout the rest of this section.



Model	Average Training Time (seconds)	Accuracy
Hidden Markov Model	0.61	93%
Decision Tree Ensemble	0.23	92%
Neural Network Classifier	3.15	90%

Figure 21: Bar chart and table plotting average training time of each model.

HMMs were our best performing model – achieving an accuracy of 93% on the testing data. I credit this success to HMMs strength in understanding sequential data. In addition, HMMs were able to achieve these results in a remarkably quick (average) time of 0.61 seconds. Also, I found that with the right displays, HMMs could be informative – especially the transition and mean matrices which were very proficiently able to describe the relationships between the features and the models.

On the other hand, I found the Neural Network Classifier extremely difficult to understand. With its black-box approach, there was extraordinarily little way of understanding how the model works. Also, our Neural Network Classifier was our worst performing model with an accuracy of 90%. While this is still a good result, our other models performed better. This is most likely due to the less complex nature of our data allowing our less complex models to excel. Furthermore, the biggest downfall of the Neural Network Classifier was the training time. It took a shockingly long time of 3.15 seconds to converge, which is 525% slower than the HMM.

Unlike the Neural Network Classifier, the Decision Tree Ensemble had a remarkably quick training time of 0.23 seconds. Its strength in training time was replicated with its performance with an impressive accuracy of 92%. This is testament to how robust Ensembles are and how good they are at counteracting overfitting. However, whilst singular Decision Trees are popular for their fantastic interpretability, visualizing the ensemble is no mean feat. Although I was unable to display to ensemble, it suffices to understand a single tree with a diagram like in Figure 6 and then imagine how it would collectively use these trees to reach a decision.

In summary, whilst all our models performed well, for this dataset the HMM and Decision Tree Ensemble excelled in performance and in a fraction of the time compared to the Neural Network Classifier.

Section 2 – Clustering and Dimensionality Reduction

This section I will use a dataset which contains information about breast cancer in Wisconsin. This dataset contains 569 datapoints – each with 30 features that we will use to determine whether a tumor is benign (B) or malignant (M).

Task 4 – PCA

Principal Component Analysis (PCA) is a statistical method to remove dimensions from a dataset whilst retaining as much variance as possible. The aim of PCA is to reduce the amount of data a model needs to classify a data point. Ideally, these features will be orthogonal to each other (i.e., they have no correlation) so that there is no overlap between the information that they contain.

To do this, I began by separating the target data from the feature data, and then used sklearn's built-in PCA function on the feature data to reduce the dataset to two principal components. By scattering the two principal components against each other and then coloring them based on their class I achieved the results in Figure 22.

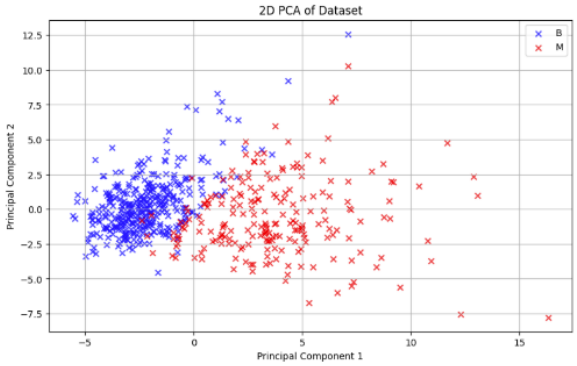


Figure 22: a scatter graph of my principal components, colored by their actual classification (see legend).

Figure 22 shows that our PCA has worked because the classifications look clustered to an extent. This means that the two principal components have successfully captured the variance in the data.

To affirm this idea, I found the explained variances of the first and second component, which were 44.3% and 19.0% respectively. This aligns with our graphical results because principal component 1 has its points less condensed meaning it captures a greater variance. Furthermore, this aligns with the fact that the first principal component captures the most information about the dataset because it has a higher variance.

Task 5 – Gaussian Mixture Models

Gaussian Mixture Models (GMMs) are a type of probabilistic model which clusters datapoints by assuming they are defined by a sum of gaussian distributions, and each cluster is itself a gaussian distribution. GMMs use soft clustering which means that a datapoint is not necessarily designated one cluster, but rather it is given a confidence score to say how likely it is that a point belongs to said cluster.

To implement this, I used sklearn's GMM and fit it using my principal components from task 4. I then got the model to predict the probability of a point being either class B (Blue) or class M (Red). This is also known as the responsibility of each component, where component 1 and 2's responsibilities are represented by red and blue respectively.

Then, I used these probabilities to weigh how blue and how red each point should be, which got me the results in Figure 23. The redder points mean that the first gaussian distribution was more responsible for this point, and the second gaussian distribution was more responsible for the bluer points. The points which are more purple are points that could not confidently be assigned to either gaussian.

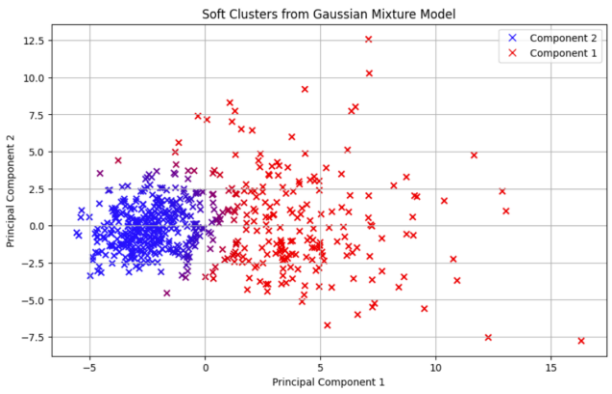


Figure 23: a graph showing a scatter of the principal components – this time colored by responsibilities in predicting a class.

Task 6 – PCA and GMM Graph comparison

In the graph from figure 22 we made a scatter graph that plots the values of principal components 1 and 2 against each other, and colors them with their associated class. This graph shows how well our principal components varied depending on their class which can be seen as there is a clear divide between our classes based on their principal components. This graph only contained two colors because it was using the ground truth class labels so we could definitively say that it was one class or the other.

Although scattering the same points, the graph in figure 23 captures different information to the graph in figure 22. The main difference in the graphs is the source of the colors. Unlike figure 22, the graph in figure 23 has multiple colors which is because the colors are based on the responsibility of each component towards each prediction. The existence of purple-colored points indicates that the GMM could not assign them to either gaussian distribution. This is an example of soft-clustering, and it is useful to see the confidence of the model on its decisions.

Similarly, to figure 22 we can see a clear divide between the two classes – but this time it is less linear as we can see a curved decision boundary. This is because PCA is trying to keep the variance at a maximum (I.e., linearly separated) whilst GMM is a probabilistic model that aims to find patterns that can be modelled by gaussians which tend to me more complex and less linear.

Task 7 – Support Vector Machines

Support Vector Machines (SVMs) are a supervised learning method for classification. It aims to use regression to find a hyperplane that best divides the data into classes.

To implement an SVM, I firstly scaled my data, and separated it into training, testing, and validation data. Then I used sklearn's SVC (support vector classification) model and fit it to the training data – using GridSearchCV to recursively find the best hyperparameters for the model.

The hyperparameters that I varied were: 'C' which is the regularization parameter, 'kernel' which is the type of kernel used in the algorithm, and 'gamma' which is a coefficient for the kernel that determines how influential each training datapoint is.

My results show that the optimal result is an accuracy of 97.3% and it is achieved when using a linear kernel, with a C value of 0.01, and a gamma value of 'scale.' Since a linear kernel achieves the best performance, it implies that our data is linearly separable. Also, lower values of 'C' means that the algorithm will be harsher on wrongly classified datapoints, which is why it got the best results. Lower values of 'gamma' mean that further away datapoints are less influential on the model. When 'gamma' is set to 'scale' it automatically adjusts the value of gamma based on the data it is given which gets the best results as it has been tailored to our dataset.

Task 8 – SVM using PCA

Now, I am going to see how SVM is affected by only using the principal components that we gathered from PCA. To begin with, I had to scale the PCA data using sklearn's

StandardScaler and then repeated the same process of finding optimal hyperparameters using GridSearchCV. I was expecting similar hyperparameters to attain the highest score, but I was surprised to see optimal hyperparameters of C = 50, gamma = 0.1, and kernel = 'rbf.'
Immediately, I was intrigued as to why every parameter saw a change. As to why the kernel changed, the model struggled to separate the points linearly like before due to the loss of dimensions making a smaller margin between the data. This can be seen in figure 24 where the decision boundary is drawn, and it is very non-linear.

Furthermore, a larger C value was probably preferred because of this a smaller margin between datapoints. This will give each point a greater influence on the decision boundary, which would explain why it is less linear. Also, the lower gamma value is probably due to the data being closer together as it means each data points influence is not reaching as far.

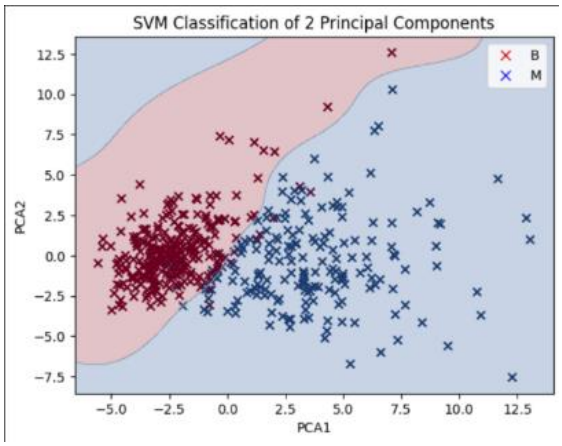


Figure 24: a graph showing the classifications of SVM using 2 principal components.

Task 9 – Effect of PCA on SVM
As can be seen in figure 25, by using more features, our model achieves a 0.7% better accuracy on the test set, in approximately twice the time. Our model will have dropped in accuracy when using PCA due to loss of information, but, due to the high explained variance of our PCA this drop was only minimal.
Whilst using more features does take more time, these amounts of time are admittedly futile for our dataset since they train so quickly. However, on more complex datasets it could be something worth considering.

Features used (with optimal hyperparameters)	Accuracy	Training time (s)
2	96.6%	0.001604
30	97.3%	0.003331

Figure 25: a table showing the accuracy and training time of SVM when using 2 and 30 features.

Section 4 – Bayesian Linear Regression

In this section I will be using the data about bike rentals in Seoul. The aim of this section is to use Bayesian Linear Regression (BLR) to produce accurate estimates for the number of rented bikes given 13 features about a specific date such as the temperature or the precipitation.

Task 10 – altering data for BLR
Before being able to do any linear regression, I needed to clean the dataset.
Firstly, some of the data contained non-numerical values which are unsuitable for linear regression. To amend this, I designated each non-numerical value with an integer id. For example, the 'Functioning Day' column contained values 'Yes' and 'No' which I replaced with 0 and 1, respectively.

Secondly, some of the features are highly correlated which makes it unnecessary to include both columns. For example, season will be highly correlated with temperature so I will exclude the season column since it has a lower variance than the temperature column.
Finally, I ended up removing variables with lower variances as they will not have the information to pick up strong patterns through regression.

Task 11 – Prior distributions
For my prior distributions, for each feature I used I set up a normal distribution with a mu I deemed to be reasonable by estimating. For example, I thought: "What temperature would be ideal for a bike ride?" and came up with a mu value of 25 degrees Celsius – repeating this for each prior distribution.
Then, I set sigma to a higher value for features I expect to have less impact. This is because with a higher sigma value, these variables should have a smaller coefficient in the model giving them less impact on the final result. However, I used a uniform distribution for sigma because I had no prior information on the dataset – so giving all sigma's an equal chance seems like the fairest way.

Task 12 – Perform BLR on new dataset
Once my priors were set up, I created a pm.Model(), and then did the dot product on my prior distributions with the corresponding X values. I added this to my intercept distribution and then found a likelihood normal distribution. Next, I used pm.NUTS() to create 1000 samples from the posterior distribution. The results of my BLR's arviz.summary() are shown in figure 25

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
Intercept	131.039	43.104	49.927	212.099	1.608	1.141	719.0	1112.0	1.01
Hour	28.042	0.773	26.577	29.434	0.017	0.012	2130.0	1401.0	1.00
Temperature	31.807	1.980	28.024	35.443	0.072	0.051	749.0	1132.0	1.01
Humidity	-4.072	0.500	-4.997	-3.125	0.020	0.014	648.0	1152.0	1.01
Wind speed	1.082	1.815	-2.417	4.370	0.038	0.036	2254.0	1389.0	1.00
Visibility	0.066	0.009	0.048	0.083	0.000	0.000	1991.0	1578.0	1.00
Dew point temperature	-3.477	2.065	-7.339	0.296	0.078	0.055	702.0	945.0	1.00
Solar Radiation	-0.218	0.991	-1.939	1.684	0.020	0.023	2451.0	1379.0	1.00
Rainfall	-33.369	3.389	-39.685	-27.100	0.064	0.046	2839.0	1364.0	1.00
Snowfall	1.054	4.589	-7.465	9.724	0.093	0.117	2470.0	1553.0	1.00
sigma	473.552	3.666	466.720	480.424	0.076	0.054	2329.0	1237.0	1.00

Figure 25: results of arviz.summary() for my BLR model

Task 13 – Evaluate BLR

I think that the summary provided by arviz.summary() shows that to an extent the MCMC sampling has generated reasonable approximations to the posterior distributions. My reasoning for this is my r_hat values have all converged to 1.00 (except for temperature and intercept with 1.01).

Task 14 – Biggest influences on bike hires

From looking at the posterior mean values, I can infer which posteriors had the most positive effect and which had the most negative effect on the bike hires. The posteriors with larger mean values had a more positive correlation with the number of bikes hired. A good example of this is with temperature which (other than intercept) got the most positive mean. Logically, this makes a lot of sense as intuitively more people would want to ride a bike when it is warmer outside.

On the contrary, our most negative value for posterior mean was with rainfall. Again, this is a very good sign as intuitively less people would want to ride a bike when it is raining – therefore it sees a negative correlation with the number of bikes rented.

As well as this, we can look at values with correlations close to zero to see values that had little to no input on the number of bikes rented. One example is with the visibility which makes sense because visibility is rarely a cause for concern when riding a bike (with exception of extreme situations). However, another example of this is with snowfall. I thought that snowfall would have more of a negative correlation with the number of rented bikes, but instead it has a tiny positive correlation. So, maybe I am an anomaly who doesn't like to ride in the snow, or perhaps the value is off because of the lack of variance in the snowfall data causing my chosen posterior to be inaccurate.

Furthermore, I expected solar radiation to have a positive correlation with the number of bikes rented because it seems extremely likely that more people would ride when it is sunny.

Task 15 – Bayesian Linear Regression Summary

In conclusion, I do not think that Bayesian linear regression was a suitable model for this kind of data. The main reason for me saying this is the extremely large sigma value for our predictions. Considering that on an average day there is probably around 500 bikes rented – to have a sigma of 474 is basically useless. From this sigma value we can essentially deduce that our model may be able to spot extreme spikes in the bike rentals, but even for moderate changes it is completely unreliable. In summary, I think in its current state, the system is too complex to be predicted by Bayesian linear regression – and it is likely that it is overfitting to the larger datapoints. To improve the performance of the BLR model, I may need to further reduce the number of features in use, and also increase the sigma values to try and reduce overfitting.