University College London

Faculty of Engineering Sciences

MSc Computer Science Summer Project

MSc Computer Science

Predictive Maintenance of Solar Panels

Jake Connolly

Supervisors:
Luke Dickens

Email:
l.dickens@ucl.ac.uk

Authors:
Jake Connolly

Authors Email:
(jake.connolly.13@ucl.ac.uk)

13th June 2018

Abstract

Abstract text found in header.tex file

Contents

1 Introductionzz

This report evaluates the performance of different regression models at predicting the quality of a variety of Portuguese wines. A dataset has been used which describes red wine via 11 different features, including alcohol content, fixed acidity and quality. The task has been to assess the efficacy of different machine learning models in predicting the quality of a wine given the other features.

This reports discusses four regression models for that were created to predict wine quality. This includes 3 linear basis function models polynomial, bayesian and radial basis as well as K-Nn. These models were selected to sample a selection of some of the most popular Linear regression models. By optimising each model, a fair comparison of each models efficacy was made using the same scoring for each.

1.1 Performance Evaluation

The metrics with which we have evaluated the performance of these models are as follows:

- Root mean square error (E_{RMS}): this is measure of prediction accuracy. RMSE disproportionately affects the points further from the actual results, therefore favouring predictions with small variance. This ensures that the metric is more sensitive to outlier values. This is the primary metric that we have used to asses our models. (E_{RMS}) is described by:
$$E_{RMS} = \sqrt{\frac{2E(\mathbf{w}^*)}{N}} \tag{1}$$
- Mean absolute error (E_M): this is a measure of the actual distance of the error from the prediction. This is less sensitive to large errors than E_{RMS}, but gives a clear measure of how far predictions on average deviate from the target.
- Mean absolute percentage error ($E_{M\%}$): presenting mean absolute error as a percentage can help gauge the performance of the model in a more conceptual way, helping to gauge how big the error is
- Median absolute error ($E_{\tilde{x}}$): This is the middle most error of predictions. This measure is useful as it is unaffected by outliers giving a picture of how the general accuracy of the model
- Variance (σ^2): this measure highlights the precision marked by the spread of predictions. A low variance indicates that prediction are all made in a similar region with fewer outliers. However a model could have some issues with it's accuracy, this is why the variance needs to be used in conjunction with a mean value.

A function named `error_score.py`, was created which analysed predictions, making sure all the models were evaluated equally; enabling direct comparisons to be made.

1.2 Cross Validation

To maximise the data that we have at our disposal we validate our models using K-fold cross validation. This separates the data into K separate folds, of which K-1 are training folds and one is the testing fold. The model is then trained and tested K times, each time using a different fold for testing. We then take the average error values of all the folds. This ensures we reduce our bias (more fitting data) and variance (more validation data) as all of the data becomes both training and testing data.

Table 1.1: Example showing data splitting for 5-Fold cross validation

	fold 1	fold 2	fold 3	fold 4	fold 5
iteration 1	test	train	train	train	train
iteration 2	train	test	train	train	train
iteration 3	train	train	test	train	train
iteration 4	train	train	train	test	train
iteration 5	train	train	train	train	test

validation phase validation

In order to validate our model on previously unseen test data we held out a validation data set. This set is 10% of the original data. This means we perform K-fold cross validation on the other 90% of the data.

2 kNN Regression

Table 2.1: kNN Regression Error Scores

E_{RMS}	E_M	$E_{\tilde{x}}$	E_{MP}	σ^2
0.7933	0.6289	0.5	11.1712	0.6280

2.1 Overview

The k-nearest neighbors regression splits data into target and training data sets. It predicts the target set by finding the average of the k-nearest values in the training set of data. It is therefore considered a 'lazy' algorithm as it does not feature a training phase and therefore doesn't generate a function which can later be tested.

$$\hat{y} \quad = \quad f(x) \quad = \quad \frac{1}{k} \sum_{x_n \in_k (x)}^{k} t_n \tag{2}$$

The distance between two inputs is calculated using the Minkowski distance function. This function takes a parameter **p** which determines whether the function takes the Euclidian (p = 1) or the Manhattan distance (p = 2) between the input rows.

$$D(x_i, x_j) \quad = \quad \left(\sum_{l=1}^{d} |x_{il} - x_{jl}|^{1/p} \right)^p \tag{3}$$

2.2 Method

- Select number of folds for cross-validation and number of **k** values to test
- Create cross-validation folds given size of data **N**
- A dictionary of matrix of errors is created to hold the different error terms for each fold iteration
- For each fold separate the data into training and target sets
- Plug the training and target data sets into the **kNN regression function**
- **If k = 1:** For each row in the target set find the Euclidian/ Manhattan distance between it and each row of the training set. Store these distances in the columns of the 2-dimensional distances matrix.

Create a matrix with corresponding training targets for each training row. Sort these two arrays based on increasing distance and store in the sort matrix

- Now there is a matrix that holds all the distances for all targets
- For each value of k take the first k values of the target column of the sort matrix and divide their sum by **k**
- This is the kNN predicted target value which stored in the kNN_targets array
- Given the actual target values and kNN values compute the error values (RMSE, Mean, Median etc.) and store them in the error matrix
- Repeat this for all values of **k**
- Return the error matrix for that iteration of cross-validation
- Repeat until there are *K* number of error matrices for the *K* number of folds
- Take the average of the *K* matrices to get an error matrix with the average errors
- Find the k value for the minimum error of each error type

2.3 Results

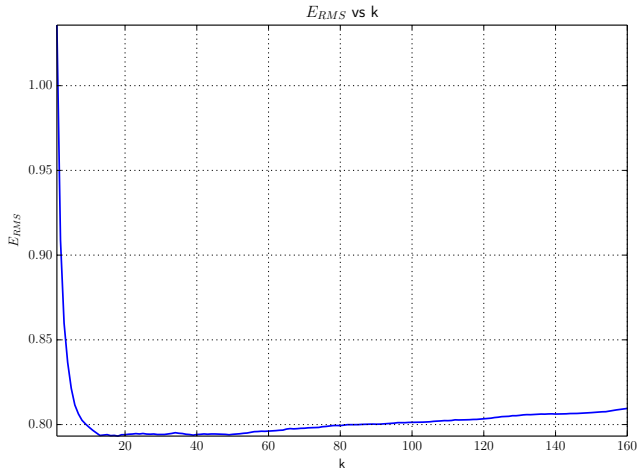

Figure 2.1: E_{RMS} Variation with k

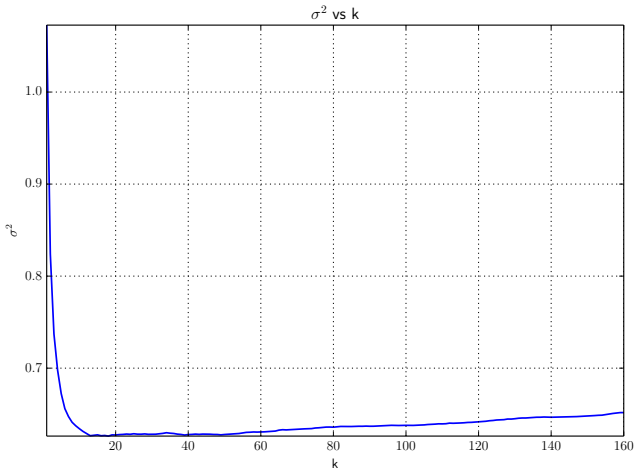

Figure 2.2: σ^2 Variation with k

2.4 Discussion

As seen on Figure 2.1 the kNN regression root mean square error is minimized at a k value of **18** using the Manhattan distance measure. This means that the method returns the most accurate results when distances are compared with the 18 nearest neighbours (Note: The k-value varies as tests are run due to the random nature of our cross-validation method). Lower values of **k** result in gross over-fitting as seen by the sharp increase in the error value.

The kNN by nature has high-variance (2.2) but that can be accounted for by increasing the number of neighbours affecting the prediction (**k**) as seen on the second graph above.

The model is simple to implement and gives immediate results but unfortunately is very resource intensive. Each time kNN is run it needs to iterate through each training row for every target then sort the data, which makes it both computionally and storage intensive (especially with large training data sets). Another major disadvantage is that it is a so-called 'lazy learner' as it does not return any generalized model which could be used for further testing.

3 Linear Basis Polynomial Regression Model

Table 3.1: Polynomial Model Error Scores With Dropped Low Scoring Features

Features	E_{RMS}	E_M	$E_{\tilde{x}}$	E_{MP}	σ^2
Dropped	0.6373	0.5035	0.4136	8.9696	0.4033
All	0.6544	0.5171	0.4353	9.2190	0.4258
% Change	2.7%	2.7%	5.2%	2.8%	5.6%

Degree Factor $= 3, \lambda = 0$

3.1 Preliminary Linear Regression Analysis

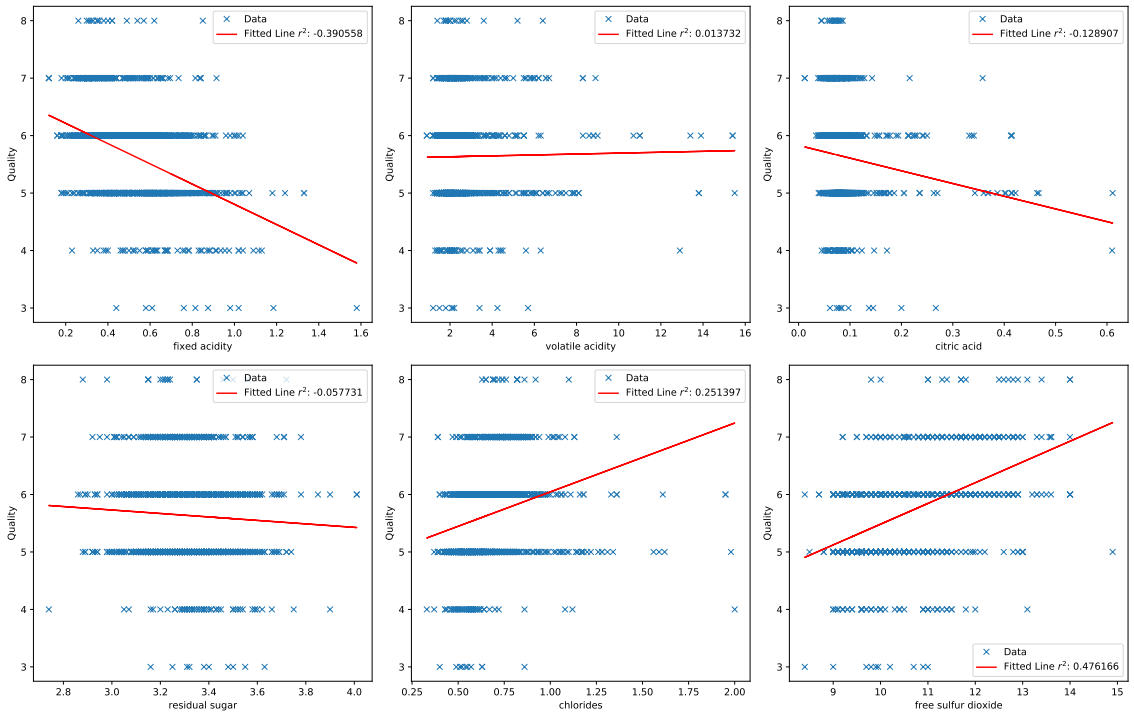

Figure 3.1: Comparison of correlation of each feature against wine quality

To gain a general understanding of the relationship between each feature and its effect on wine quality, scatter plots comparing each feature were created; providing a simple understanding of the data relationship. Through calculating the regression values between features and the label (wine quality), parameters which may distort a linear regression model were highlighted. Figure 3.1 gives an overview of this some of the strongest and weakest relationships. It is worth noting that pair wise correlation cannot be visualised using this method, meaning simply dropping features of low correlation may lead to decreased model performance.

3.2 Overview

A linear basis model is a method of supervised machine learning, that takes a series of features and assumes that a (basis) function can be applied to these features to predict a target [1]. These basis

functions are summed together to create a linear combination of function, given it the *linear* name. A linear model can be generalised as the following:

$$y(x) = w_0 \quad + \quad w_1\phi_1(x_1) \quad + \quad w_2\phi_2(x_2) + ... + w_n\phi_n(x_n) \tag{4}$$

$$where \quad \phi_n(x_n) = n \text{ basis functions} \tag{5}$$

Equation 5 is a generalisation of linear regression that essentially replaces each input with a function of the input. In the case where the basis function is the identity matrix, the model becomes just linear regression. The type of basis functions (i.e. the function ϕ) is chosen to model the non-linearity in the relationship between the inputs and targets [2].

For this model, a polynomial basis function evaluated, that can be defined as:

$$\phi_j(x) = x_j \tag{6}$$

Different polynomial models can be tested by varying the degree factor (j). The degree factor could be optimised by looping through a range to find the factor which minimised E_{RMS}. It is worth noting that at a factor of 1, the polynomial model is evaluating a basic linear regression model. As a second factor to optimise, the model ridge regression was applied to the model's weights. The regularisation coefficient is used to penalise weights terms with large values, smoothing out the curve of by reducing peaks.

After running the model for all methods, some experimentation into dropping certain features was made to see if the prediction could be improved. Analysing figure 3.1 the 3 features with the weakest relationship were dropped from the model.

3.3 Method

- Use the `cross_validation` function to reserve 10% of the data for the validation phase. This 10% will test the model's performance on unseen data after all training and testing is complete
- Split the remaining data across a series of 10 folds to cross-validate the model using the `cv_folds` function
- Each fold is run through the polynomial training model using the function `cv_evaluation_poly_model`, this extracts the data in each fold splitting into training, and testing sets runs the polynomial model and stores the results in a dictionary for later evaluation
- The `polynomial` function, takes training and testing data iterates through all degree factors within a range, and then each regression coefficient to find the optimum settings for the polynomial model. To find the ranges of both variables a variety of well-spread points were picked to test the model's sensitivity; refining these ranges to capture the minima in results. Ranges of $1-5$ (degrees) and $\lambda(0-51)$ were chosen. The `polynomial` function can be further broken down into the following steps:
  1. `expand_to_2Dmonomials` takes the input matrix and expands this expands out the columns of the input matrix multiplying by the factorial of the polynomial degree. i.e. for a degree factor of 2 the matrix gets expanded from x to $1, x, x^2$
  2. The function `regularised_ml_weights` is then used to create a unique weight for every polynomial factor (column) in the matrix; i.e. a degree factor of 2 with 11 features would be converted into 23 different weights. The function also regularises the weights, subtracting λ using

equation 9; helping smooth out the function. A value of $\lambda = 0$ is equivalent to no regularisation coefficient

3. `construct_3dpoly` sets a prediction function taking the inputs of the regularised weights and the degree factors for the polynomial basis function
4. (`prediction_function`) is used to create a target prediction for a series of inputs based on the previous parameters of the model. `expand_to_2Dmonomials` is reused to multiply each value by the basis function x^j
5. Finally `polynomial` collects the error values using the `error_score` function and the weights analysing the performance of the model

- For all data folds, repeat steps $1 - 6$. After taking all the different cross-validation results from the model; these are aggregated to produce a final error score for the model
- The weights corresponding to the optimum degree factor and λ are aggregated to create a final model for validation.
- The final model is given new testing data (reserved at the start of the method) to validate the model efficacy

3.4 Results

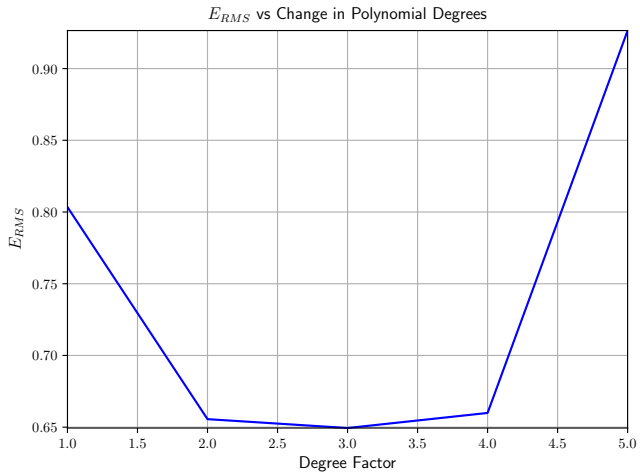

Figure 3.2: E_{RMS} Variation with Change in Polynomial Degree, $\lambda = 0$

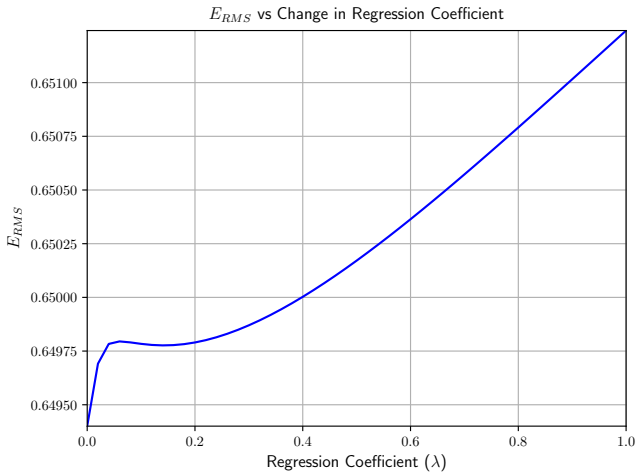

Figure 3.3: E_{RMS} Variation with Regression Coefficient (λ) where $Degree = 3$

3.4.1 Discussion

Figures 3.2 and 3.3 both describe a snapshot of the sensitivity which the polynomial model has to changes in the degree factor and λ; this is because the model iterates through every combination of degree and λ to find the perfect combination. On first observation of the the performance of the Polynomial model, it was witnessed that there was a significant improvement in accuracy when using a polynomial basis function over a basic linear function on the wine data set. Figure 3.2 shows a parabolic shape that reaches its minimum point at a degree factor of 3 with a slight increase in error for both 2 and 4. The model's sensitivity between 2 and 4 can also be observed when comparing the optimum models across different folds. Both the model's variables alter these three values and their corresponding optimum regression coefficient. The models sensitivity therefore eludes to requiring more data required generates a stable prediction across the folds.

Through holding out data in the validation phase, overfitting could be avoided due to comparing the model to data which has been included in the training phase. The prediction made in during the validation test is slightly worse than the value observed during the validation phase. This again is likely due to the variance in the data.

By using the regression relationship between a feature and its effect on quality highlighted in figure 3.1, the features residual sugar, free sulphur dioxide and pH were dropped all having an $|r^2| < 0.1$. Dropping more features appeared to have a negative effect on the models prediction. Originally, the polynomial model would select a value of $\lambda = 0.22$, suggesting that the model was overfitting to some of the data-points. By removing these features λ dropped to 0, suggesting the model was naturally fitting the data more smoothly. Table 3.1 shows the positive effect that dropping these features has on the model.

4 Bayesian Polynomial Regression Model

Table 4.1: Bayesian Polynomial Model Error Scores

E_{RMS}	E_M	$E_{\tilde{x}}$	E_{MP}	σ^2
0.6423	0.4984	0.4036	8.8329	0.4111

4.1 Overview

A Bayesian Model for linear regression aims to reduce the effect of over fitting, as well as provide a measure of the predictive distribution of the model. This is achieved by providing a prior probability distribution over the model parameters. The mean and covariance of this distribution along with the maximum likelihood estimates arrived at by linear model is used to calculate the posterior distributions. The posterior mean and covariance are given by the following equations respectively.

$$m_n = S_N(S_0^{-1}m_0 + \beta\Phi^T t) \tag{7}$$

$$S_n^{-1} = S_0^{-1} + \beta\Phi^T\Phi \tag{8}$$

The posterior covariance is used to derive a the predictive distribution for a given input, provided by the equation

$$\sigma_N^2(x) = \frac{1}{\beta} + \Phi(x)^T S_N \Phi(x) \tag{9}$$

The predictive distribution can be used to compute the upper and lower bounds of the predicted values generated by the mean posterior weights. This can provide a graphical representation of the certainty of the model.

For the implementation of our bayesian model we used a polynomial model as the basis function to which it is applied. As the cubic polynomial model was the most accurate of our non-bayesian models, it was the natural choice.

4.2 Method

- Select number of folds for cross-validation
- Create cross-validation folds given size of data N
- A dictionary of matrix of errors is created to hold the different error terms for each fold iteration

- For each fold separate the data into training and target sets
- Train and test inputs and targets are passed as parameters to the `polynomial_bayesian` function.
- The polynomial degrees and regularisation coefficient are given the optimum values ascertained by the polynomial model described in section 3
- The `expand_to_2Dmonomials` function takes the training input data matrix and applies a polynomial basis function to the values. this returns an output matrix of each input value expanded within the range given by the polynomial degree value
- A prior mean weight, **m0** of 0 and an α of 100 are given
- The precision parameter β is deduced by finding the variance of the raw data
- `calculate_weights_posterior` takes the output matrix training targets, β , prior mean weights and covariance and returns the posterior mean weights and covariance
- Within a loop, these posterior values are then returned as prior value to the `calculate_weights_posterior` and used by the `construct_3dpoly` function to generate prediction values and predictive distribution values
- Error values for each iteration are generated by the `error_score` function
- The prediction and predictive distribution values returned on the final iteration are used to give the upper and lower bounds of the predicted values
- For all data folds, repeat steps $1-6$. After taking all the different cross-validation results from the model; these are aggregated to produce a final error score, prediction values, upper and lower bounds for the model

4.3 Results

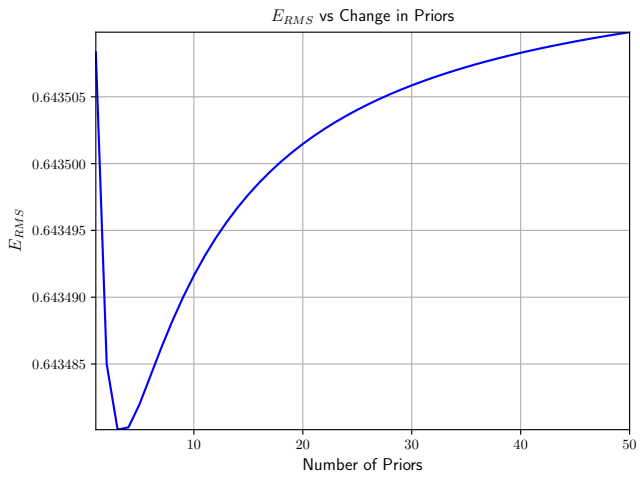

Figure 4.1: E_{RMS} Variation iterating calculated posterior values as priors

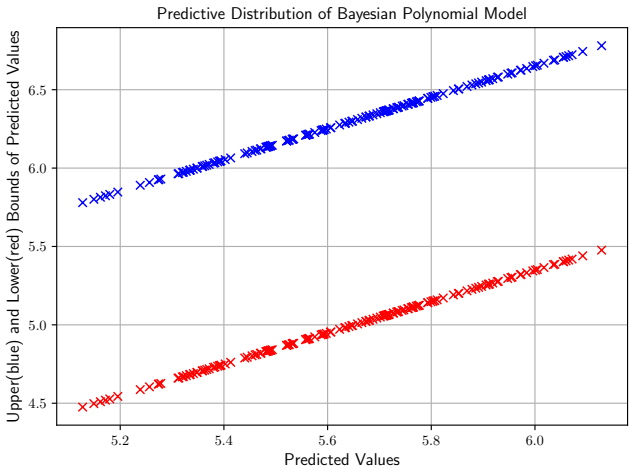

Figure 4.2: Predictive distribution of values, plots upper bounds (blue), lower bounds (red) against predicted values

4.3.1 Discussion

As you can see from figure **??** the number of iterations through the `calculate_posterior_weights` function (where each posterior is then used as the prior in the next recursion) negatively effects the accuracy of the model. This implies that, although the accuracy of the polynomial model is improved marginally by the application of a prior, continual iterations generate weights that predict values that are farther from the actual target values.

This is particularly relevant when choosing our method of deriving a precision value. The precision value β used in our final model was that derived from the variance of the raw target values. This is due to the other method of finding the β being to find the variance of the residuals. This is dependent on predicted values that themselves are derived from posterior weights, therefore a β value derived from them will become increasingly worse.

Figure **??** shows the predictive distribution of the model. This shows a largely constant σ for the prediction value, consistent with the dense data that we received. There is some slight divergence at either end of the scale of values, but we can essentially say that we are equally confident of all our prediction values.

Although removing features does increase the accuracy of the Bayesian Model to a significant extent as discussed above, this is not the case for the Bayesian Polynomial Regression model. Fewer features incorporated into the model reduces the problem of overfitting. This is reflected in the increased accuracy of the Polynomial Regression model, and is also the reason why there is no such improvement in the Bayesian model, as such over-fitting effects are mitigated already.

The primary disadvantage of using the Bayesian approach lies in that it is always uncertain what value to choose as a prior. In our model we chose 0 as this is the standard in the literature when it is not obvious where the value should lie. This can result in final predictions being heavily influenced by the choice of prior

5 Evaluation

Table 5.1: Final Results Comparison

	E_{RMS}	E_M	$E_{\tilde{x}}$	E_{MP}	σ^2
kNN	0.7933	0.6289	11.1712	0.5	0.6280
Polynomial	0.6544	0.5171	9.2190	0.4353	0.4258
Bayesian	0.6423	0.4984	8.8329	0.4036	0.4111

5.1 Comparison of Models

In order to fairly compare the performance of the different models Table 5.1 describes the performance of our models in each of the five evaluation metrics. The following conclusions can be drawn from this reports analysis:

A Polynomial Regression model is far superior to the kNN. A kNN is simple to implement but does not return a model and is very resource intensive taking around twice the runtime, therefore were the dataset to grow in size, it may not be computationally feasible. As well as this the kNN model suffers from high variance issues, although it also has a low bias

By varying the degrees of the polynomial basis function, a broad range of functions can be formulated from the data to make predictions. If there is any curvature in the data (as opposed to a step-like relationship by kNN) then a polynomial regression model is better suited to capturing it than the kNN model. However, linear models can suffer from the over-fitting of data. Some methods to mitigate this effect include: - Removing features with minimal correlation to the target, improving the model by 2.7% - Holding out data on which to validate the model, - applying a Bayesian approach by applying prior

values for the mean weights and covariance. Applying a Bayesian approach prevents over-fitting by drawing the data back to the targets generated by the initial prior values. As such, bayesian approach performs best on our E_{RMS} metric of all our models, as it prevents our prediction values from sticking too closely to the curve generated by the polynomial basis function. Another means of reducing this over-fitting is to reduce the number of features. As mentioned above, if this is implemented it increases the accuracy of the model beyond that of the Bayesian Polynomial model approach. This is conclusion is arrived at by using the E_{RMS} value as our primary error metric, however if the percentage mean and median error rate are used then the Baysesian Polynomial remains the best model.

From this we can deduce that although the Bayesian Polynomial regression model performs best when we include all the features within the dataset, if we strategically remove features from the Polynomial Regression model then that model achieves the best accuracy.

References

[1] C. M. Bishop, 'Machine learning and pattern recognition', *Information Science and Statistics. Springer, Heidelberg*, 2006.

[2] E. Cai, *Machine learning lesson of the day – introduction to linear basis function models | statsblogs.com | all about statistics*, http://www.statsblogs.com/2014/03/11/machine-learning-lesson-of-the-day-introduction-to-linear-basis-function-models/, (Accessed on 02/25/2018).