Project 1 Report: Polynomial Curve-Fitting Regression for Working-Age Data

Jordan S-H

https://github.com/jstebner/481-P1

[0] Overview

The aim of this project is to implement all components necessary to perform generalized linear regression on the data found in the '**data/**' directory.

The implementations for all models and transformers can be found in the '**src/impls.py**' file, and the usage of implementations is done in the '**src/main.py**' file which can be run with:

```
$ python3 ./main.py
```

Note that the main file can be run from any location as it can locate the positions of supporting files.

All figures used in this document and data collected from testing the model are also stored in the '**out/**' directory with names indicative of their content.

The model to be tested (in '**src/main.py**') is given as follows:

```
model = OutputScalingWrapper(
    Pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=d*),
        LinearRegressor()
    )
)
```

A rough outline of how the model seeks to perform polynomial curve fitting is given as follows:

1. The target vector is scaled prior to fitting the model (for prediction accuracy).
2. The input vector is scaled and transformed with Polynomial Features.
3. A linear regressor is fit using the transformed input and target vectors.
4. When creating predictions, the model applies the weights created when predicting to the new transformed input (using the same transformations as the training data) and un-scales the resulting vector (using the same transformations as the training target vector).

[1] 6-Fold Cross Validation for varying degrees

6-fold cross validation will be used to select an optimal degree (d*) by evaluating the average RMSE† across each fold for each d ∈ [0, 12]. The splits made are done in order (fold 1 validation set uses indices [0, 7), fold 2 uses indices [7, 14), ...) without shuffling the data as was specified in the project description (and for reproducibility of results).

The data for each degree and its associated average RMSE can be found in '**out/cv_errors_data.dat**', and the loss curve to select d* is given below:

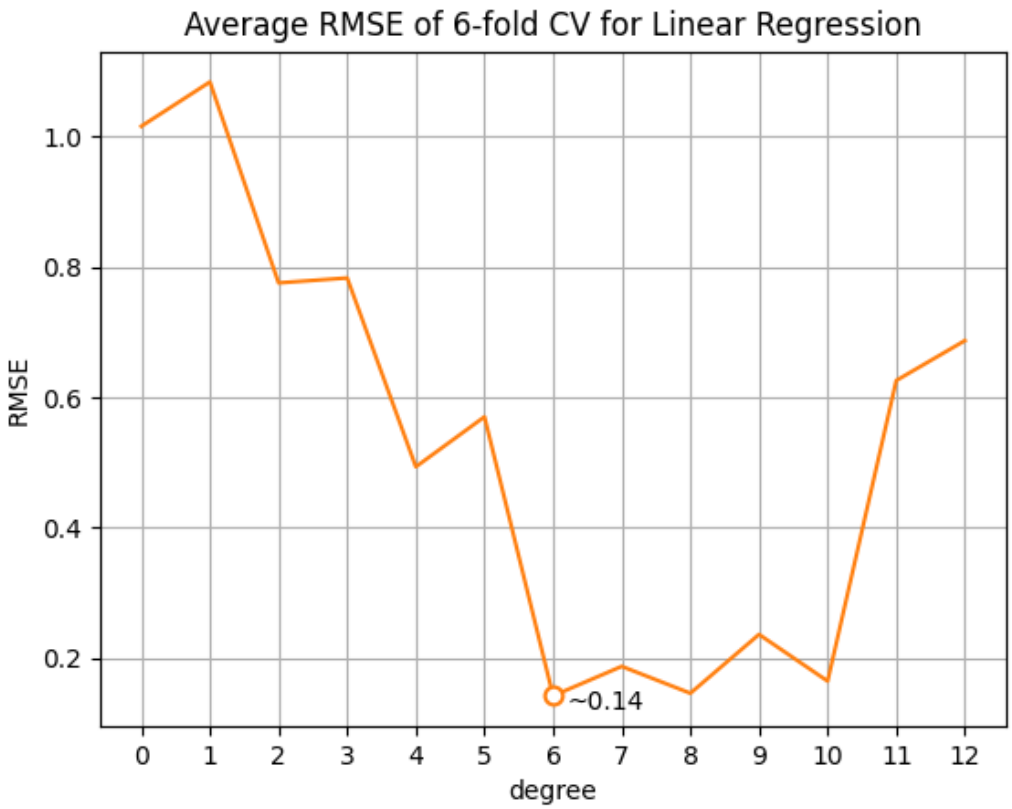

Figure 1.1

[2] Optimal degree of model - d*

Figure 1.1 shows the loss decline as model complexity increases, until it begins to rise again at higher degrees (which suggests overfitting). Also shown in the figure is the degree with minimal error, which is degree 6 with a loss of roughly 0.14. Degree 6 will be used for d* moving forward.

† See appendix 8.1 for details on implementation

[3] Weights obtained using d*

The weights obtained when fitting a model with degree 6 on all training data are stored in '**out/w_linear.dat**', and the formula for predicting output values (with rounded weights, and assuming $x \in \mathbb{R}$) is given below:

$$f(x) \approx -0.113 + 0.357x + 3.370x^2 + 0.136x^3 - 2.946x^4 + 0.008x^5 + 0.538x^6$$

Since the training data wasn't scaled after transforming with polynomial features, the generated weights can't be compared relative to each other, but further work can be done to create more interpretable weights[†].

[4] Scores of the selected model

When evaluating the model with d* on the training and testing datasets, the following RMSE values (loss values) are produced:

```
Training Loss: 0.10540106673270466
Testing Loss:  0.11432570919500114
```

** These values can also be seen when running '**src/main.py**'*

While the testing error is still greater than the training error (which is to be expected), it's surprisingly close[‡] (which was unexpected).

My interpretation of this result is that the training RMSE is the expected loss for any data consistent with the trend present in the training data set, and given that the relative error percent of the testing RMSE from the training RMSE is:

$$RE(\%) = \frac{|\text{ measured} - \text{expected }|}{\text{expected}} * 100\% = \frac{|\text{ TestLoss} - \text{TrainLoss }|}{\text{TrainLoss}} * 100\% \approx \mathbf{8.47}\%$$

One could (and I will) see this value as a **generalization error**, meaning this model was ~8.5% away from the expected loss from the training set (though this view may be naïve). I believe this interpretation is at least somewhat valid, as a test loss lower than the associated training loss could be due to a small test set or luck (though this view may also be naïve).

[†] See addendum 7.2 for further analysis.

[‡] See addendum 7.1 for details and further analysis.

[5] Prediction Curve

Below is the graph of the model fit using all training data transformed with degree 6 polynomial features superimposed on all data in the project.

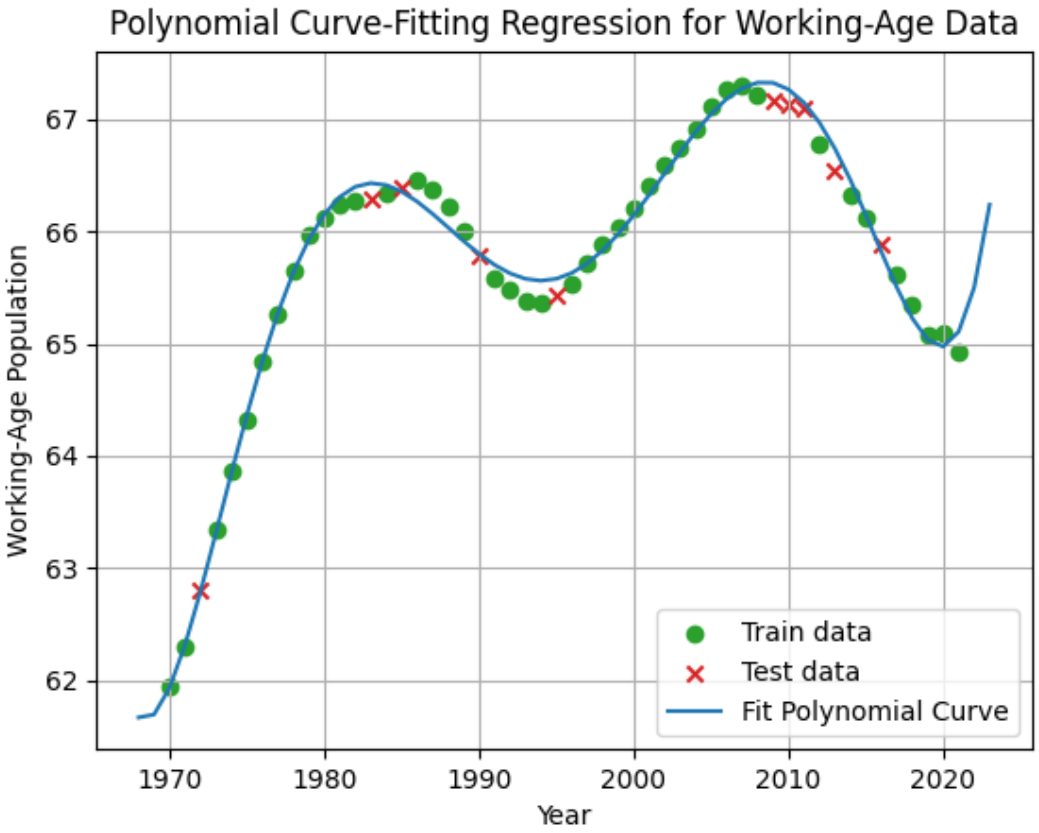

Figure 5.1

We can see in figure 5.1 that the model is able to fit a polynomial curve well to the data. There are places in the curve where it may have benefitted from a slightly higher degree (namely the peaks of the data). This may be supported by figure 1.1 where degree 8 is comparable in accuracy to degree 6 in cross validation, and further exploration could be conducted to test this claim[†].

[†] See addendum 7.1 for further exploration

[6] Reflection

I'm happy with the curve my program was able to fit shown in figure 5.1. The results discussed in section 4 also indicate that the program was able to generalize (roughly) to the training data with how low and close the training and testing losses were, though this could be due to the size of the dataset. I might want to try this model on a larger dataset to see if my program can perform as well on more/higher dimension data and to see if my implementation of polynomial features works on data with more than one feature[†].

I would also like to try testing a ridge regression model, as linear regression is equivalent to ridge regression parameterized by $\lambda=0$, meaning the results may be improved with different values of λ[‡]. An added hyperparameter to tune also gives an additional "degree of freedom" for testing, which could be fun to explore and create pretty visualizations for analysis.

At this point, the formal report for the project concludes; however, the following sections contain additional exploration of the model discussed (as well as one built using ridge regression) and technical details on motivations for how some aspects of the project were implemented (i.e., they were added for completeness and don't necessarily need to be read).

Figure 6.1 is unrelated to what was being discussed, but it's pretty, so I included it here.

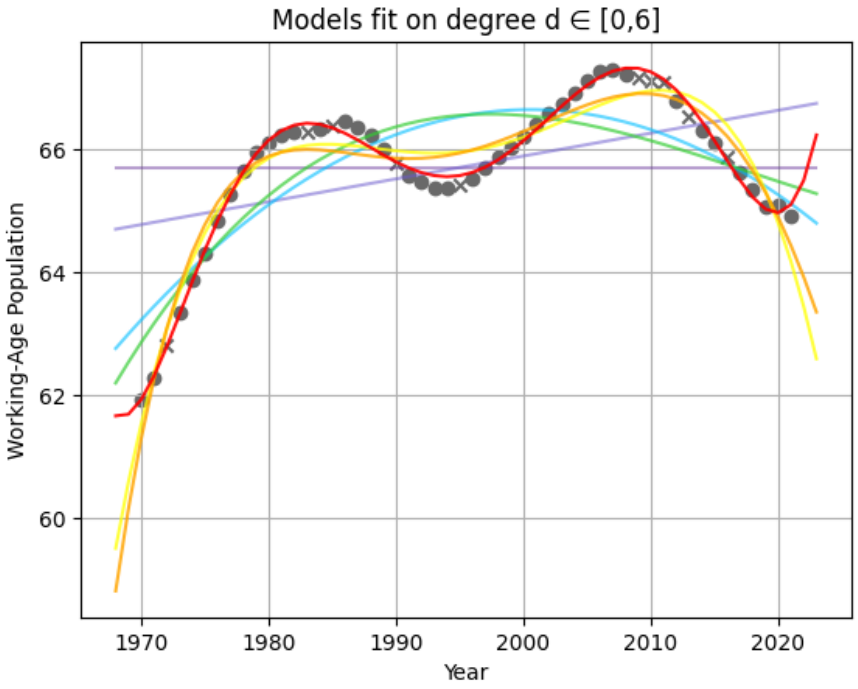

Figure 6.1

[†] See addendum 7.3 for further analysis on datasets, and appendix 8.2 for details on the implementation of polynomial feature creation.

[‡] See addendum 7.4 for details and further analysis.

[7] Addendum

** Code for this section can be found in '**src/addendum.ipynb**'*

7.1 – Test error across degrees

I wanted to see how varying the complexity of the model effects the testing loss and found that the testing error continues to decline even while the cross-validation loss starts to increase, as can be seen in the figure below. Intuitively, I feel like the test loss curve should more closely follow the trend present in the CV loss curve, and perhaps the size of the dataset influenced this difference.

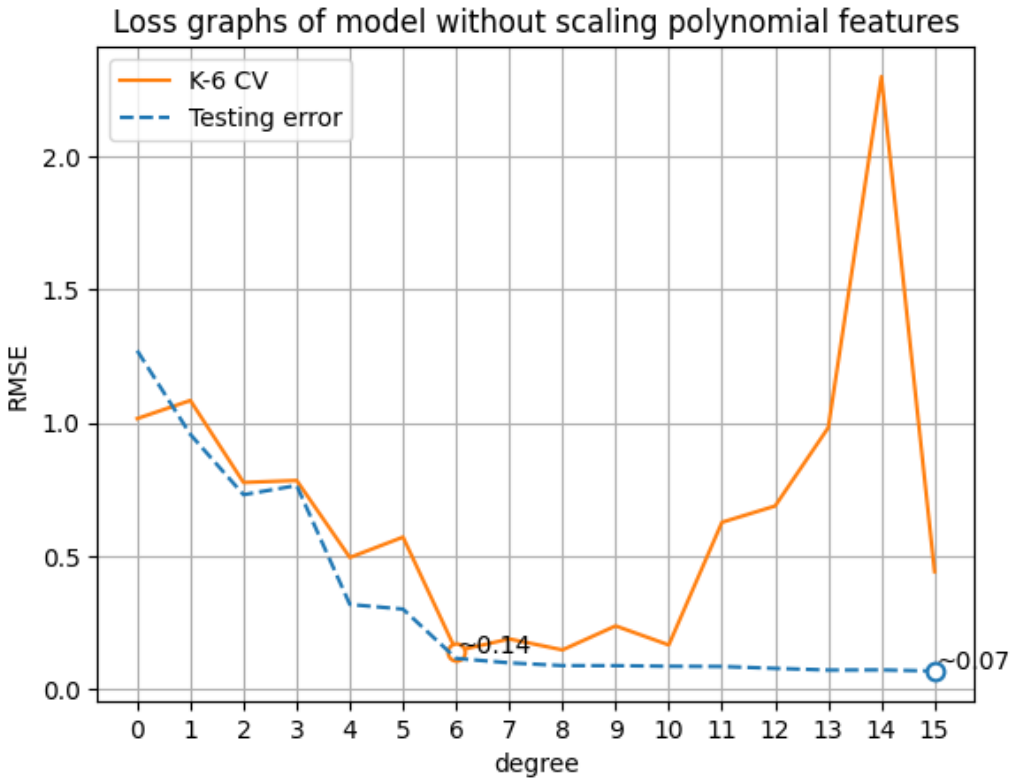

Figure 7.1.1

One thing of note is that the degree associated with the smallest CV loss is also a "hinge point" for the testing loss curve, where there are diminishing gains in accuracy in the model with increasing complexity, which could be seen as more favorable than a model with higher accuracy (as a less complex model would require less resources to work with, and the weights would be easier to interpret). This notion may be covered by ridge regression, which will be tested later.

Another thing to note from figure 7.1 is the aversion cross-validation has for degree 14 of this model, which I found funny.

I won't show it here, but I also found that the test loss jumps **heavily** at degree 33 (RMSE of +5000) which I found interesting.

In figure 5.1, I noticed that there were points of the graph that the model couldn't capture at degree 6 (namely at peaks and troughs in the data), so below is a graph of the model using degree 8 polynomial features, as figure 1.1 showed that the CV loss of degree 8 was comparable to degree 6.

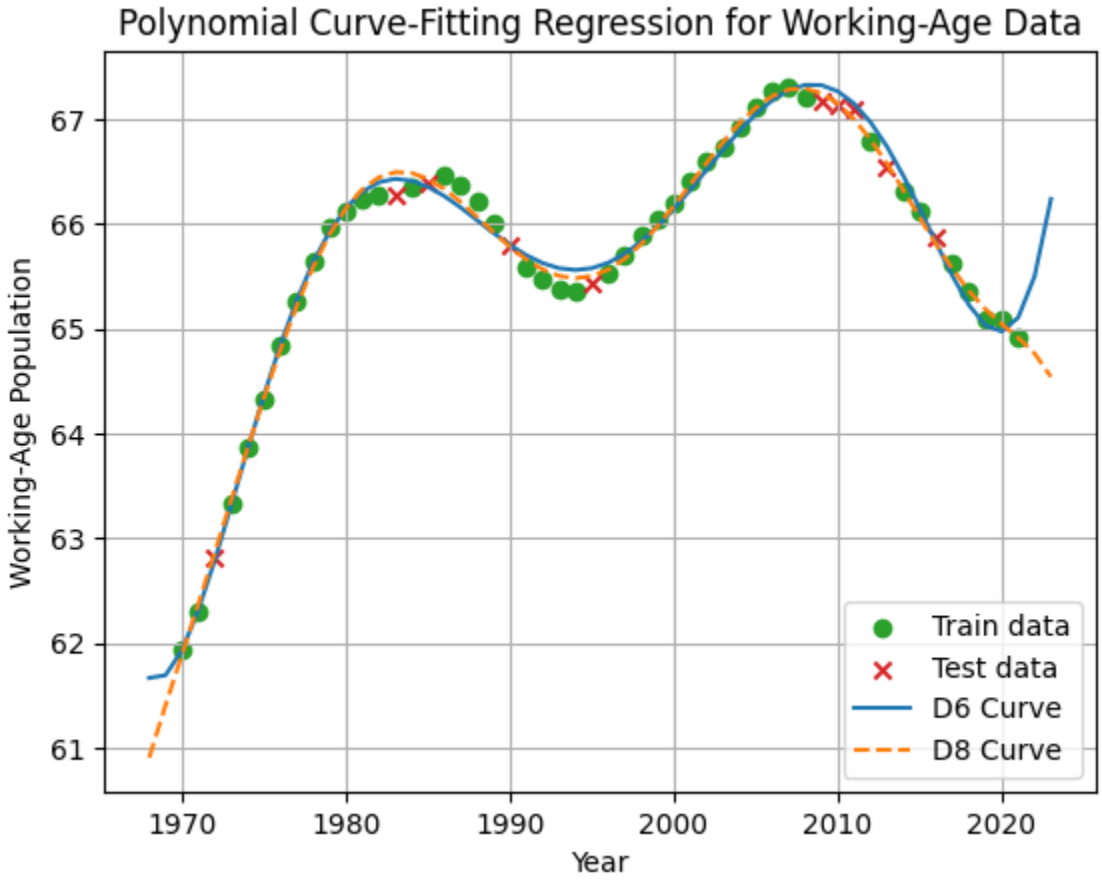

Figure 7.1.2

Degree 8 (orange dashed line) was able to fit the curve slightly closer to the data (which we knew from figure 7.1.1) since it's closer to the data around the local extrema (except for the one around year 1983), but it also appears to "better guess" the trend outside of the data that is available to it at the tails of the graph (assuming the data continues in that manner, which I have no basis for claiming).

Below are the absolute differences between the weights for each model (with the missing weights for degree 6 assumed to be 0):

0.066, 0.0677, 0.7765, 0.2193, 1.4507, 0.126, 0.8379, 0.018, 0.1487

Some of the differences are a bit higher than I expected, especially since the last 2 weights for degree 8 aren't that far from 0. It should also be restated that the effect of the terms they weight aren't greater than other terms (even if their degrees are higher) since we scale each transformed term before fitting the regressor.

7.2 – Changes in architecture

Because the scale of the input isn't necessary for the accuracy of the model, I wanted to try removing the initial scaling of the input. Below is a model without initial input scaling to be tested:

```
model = OutputScalingWrapper(
    Pipeline(
        PolynomialFeatures(degree=d*),
        LinearRegressor()
    )
)
```

Interestingly, I couldn't fit this model on the training data. The program raises a "Singular Matrix" error, presumably when calculating the Moore-Penrose pseudo inverse. The cause of this is most likely to be the discretization error when representing weights discussed in lecture which motivated the discussion of scaling, which was neat to run into.

Another change in architecture I wanted to explore was removing the output scaling since it also didn't feel necessary (though I may be very wrong like the previous assumption I made). Below is a model without output scaling:

```
model = Pipeline(
    StandardScaler(),
    PolynomialFeatures(degree=d*),
    LinearRegressor()
)
```

With this change, we can see in figure 7.2.1 that the cross-validation losses for the original model and one without output scaling perform identically (at least up to degree 15).

This result is interesting since the models are fit with different target vectors, and yet return the same results when inverse scaling the predictions of the original model.

This will be covered again when exploring ridge regression in section 7.4.

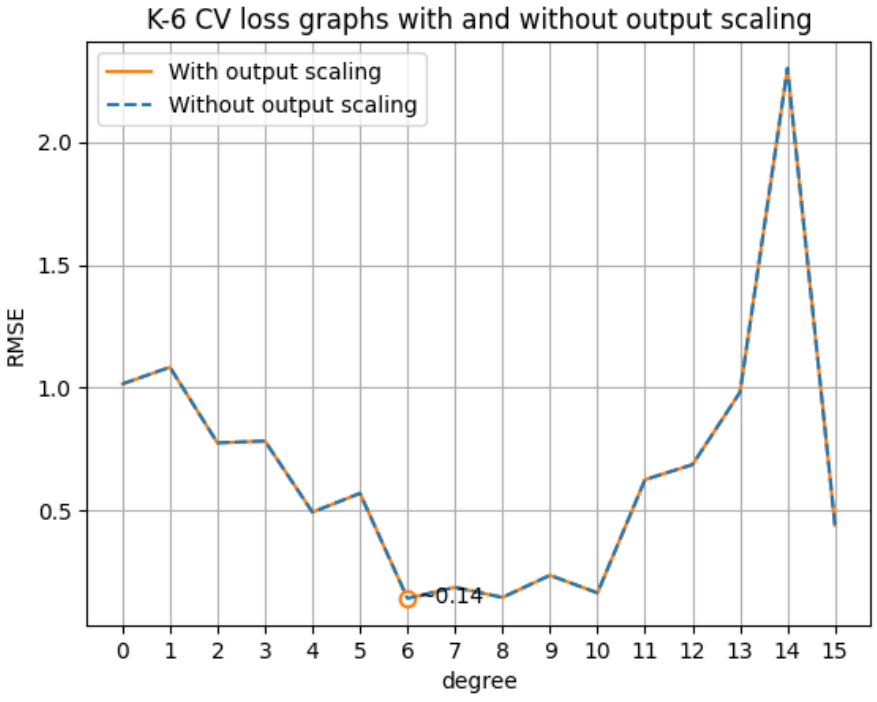

Figure 7.2.1

One final change I wanted to try was adding scaling of the input vector after transforming the input with polynomial features, as is shown in the model below:

```
model = OutputScalingWrapper(
    Pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=d*),
        StandardScaler(),
        LinearRegressor()
    )
)
```

The purpose of this model is to create weights that can be compared relatively to each other.

When fitting this model, my program ran into an issue where the linear regressor was unable to fit using the transformed training data because the determinant of X^TX for the transformed data was 0, meaning the model couldn't calculate the Pearson-Moore pseudo inverse (which can be found in section 7.2 of '**src/addendum.ipynb**').

I verified that the transformations are equivalent to those from the scikit-learn library, so the problem lies in the implementation for linear regression. To continue with exploration of this model, I'll use the scikit-learn implementation of linear regression (specifically ridge regression with λ=0).

After fitting the above model, the coefficients for predicting are:

```
0.0, 0.416, 3.672, 0.324, -8.797, 0.049, 4.393
```

** These values can also be seen in the outputs from '**src/addendum.ipynb**'*

An interesting thing about these weights is that the even coefficients (excluding 0) are weighted higher than the odd coefficients.

7.3 – Larger dataset

For this section, we'll be evaluating the model on the iris dataset from the scikit-learn library. The process for evaluating the model on this dataset will be the same as the previous evaluation.

The features of the dataset are:

```
( x₀              x₁              x₂              x₃           )
[ sepal length,   sepal width,    petal length,   petal width ]
```

** All units are in cm, which isn't entirely relevant but still worth noting that they're consistent.*

There was an included target feature, however the feature was categorical which isn't currently supported by the models studied in this project. Instead, we'll try to use the first 3 features (sepal length, sepal width, and petal length) to predict the 4th feature, petal width.

Below is the correlation matrix for intuition on the linear trends present in the data set:

```
Correlation Matrix:
[[ 1.         -0.11756978  0.87175378  0.81794113]
 [-0.11756978  1.         -0.4284401  -0.36612593]
 [ 0.87175378 -0.4284401   1.          0.96286543]
 [ 0.81794113 -0.36612593  0.96286543  1.        ]]
```

We can see that the petal width is highly correlated with the sepal length and petal length but doesn't have a (strong) linear relationship with the sepal width. We can also see there is correlation between the intended input features, but this won't be accounted for when scaling.

This suggests that the optimal degree d* of polynomial features for this dataset would likely be low but perhaps not exactly 1 (if there is a non-linear relationship between sepal width and petal width).

Figure 7.3.1 "confirms" the above statement where the optimal degree is 2 with a loss of ~0.12.

Only degrees up to 5 are shown since the loss grows very quickly and makes it hard to see the curve around the minima.

After fitting the model with all training data and d*, the following losses were obtained:

```
Training RMSE: ~0.115
Testing RMSE:  ~0.329
```

These results seem decent.

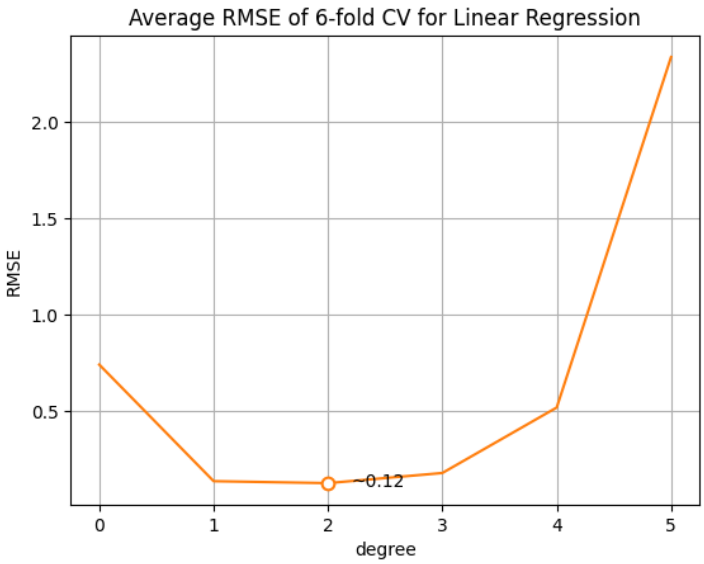

Figure 7.3.1

7.4 – Ridge regression

The following model will be used for creating and fitting a ridge regressor model to the given data:

```
model = OutputScalingWrapper(
    Pipeline(
        StandardScaler(),
        PolynomialFeatures(degree=d*),
        StandardScaler(),
        RidgeRegressor(λ= λ*)
    )
)
```

The analysis of d* for ridge regression on λ=0 will be skipped, as ridge regression on λ=0 is equivalent to linear regression†, which was covered by sections ∈ [1,6].

λ* will be selected through cross-validation of the model using degree 12 polynomial transformation. The following figure shows the loss from varying λ:

From figure 7.4.1 we can see that the optimal regularization term for 12-degree ridge regression is e^{-3}. *(The scale of the x-axis of the graph is logarithmic, and the curve continues leftward to -∞)*

The weights for the model fit on all training data is stored in '**out/w_ridge.dat**', and the train and test losses are given below:

	d*	d:12, λ*
Train RMSE:	~0.105	~0.128
Test RMSE :	~0.114	~0.129

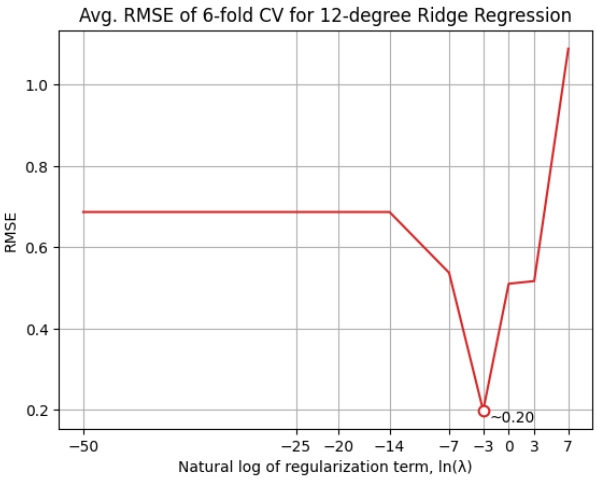

Figure 7.4.1

Figure 7.4.2 shows the curves fit using d* and λ* with degree 12. We can see that the curves behave similarly, with differences around extrema and the tails of the graph.

The training and test losses are also slightly higher for the regularized curve than the losses obtained without regularization, as depicted above (and which we can slightly see in figure 7.4.2)

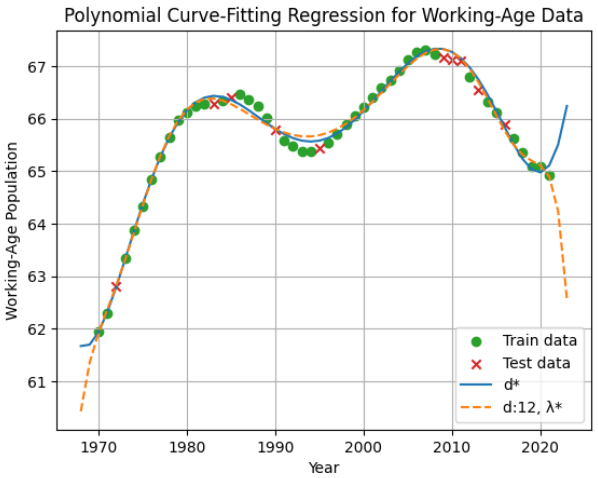

Figure 7.4.2

† Proof will be left as an exercise for the reader.

[8] Appendix

8.1 – RMSE implementation

The formula used in the RMSE implementation for this project is:

$$RMSE(a,b) \equiv \sqrt{\frac{1}{n}}\|a-b\|_2$$

I used this formula instead of calculating RMSE using its definition because the python library used for linear algebra operations is (probably) more optimized than any implementation which iterates over each vector performing the necessary calculations that I could write (and I find the code for this formula more aesthetically pleasing). The proof on why the above formula is true is as follows:

Given that:

$$RMSE(a,b) \coloneqq \sqrt{\frac{1}{n}\sum_{i=1}^{n}(a_i-b_i)^2} \quad \forall\, a,b \in \mathbb{R}^n$$

The n^{-1} term can be moved out of the square root to form:

$$= \sqrt{\frac{1}{n}}\sqrt{\sum_{i=1}^{n}(a_i-b_i)^2}$$

(Eq. 1)

Next, given that:

$$\|c\|_2 \coloneqq \sqrt{\sum_{i=1}^{m}c_i^2} \quad \forall\, c \in \mathbb{R}^m$$

We can multiply a term to each side to form:

$$\sqrt{\frac{1}{m}}\,\|c\|_2 \coloneqq \sqrt{\frac{1}{m}}\sqrt{\sum_{i=1}^{m}c_i^2}$$

(Eq. 2)

Let c = a – b. This would imply that:

$$m = n \quad and \quad c_i = a_i - b_i \quad \forall\, i \in [\,1, n\,] \cap \mathbb{Z}$$

We can use these equalities to substitute terms in equation 2 to form:

$$\sqrt{\frac{1}{n}}\,\|c\|_2 \coloneqq \sqrt{\frac{1}{n}}\sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

Which is equivalent to equation 1. We can substitute c for (a – b) to yield:

$$\therefore\ RMSE(a,b) \equiv \sqrt{\frac{1}{n}}\|a-b\|_2 \quad \forall a, b \in \mathbb{R}^n \quad 😊$$

It's a simple proof, but I felt that it would be a good exercise to work through and wanted to justify my decision for how RMSE was implemented; as well, using a "closed form" would have the added benefit of a (nominal) increase in efficiency over the iterative form (though this is yet to be tested, and claiming the norm function to be in closed form is debatable at best). *I haven't had any formal instruction on writing proofs, so please let me know if there are any ways I can improve.*

8.2 – Polynomial features implementation

The implementation for polynomial features follows the fit/transform pattern used throughout scikit-learn to stay consistent with common practice, so the fit method creates and stores a list of exponents for each transformed feature which is used by the transform method to calculate each transformed feature using the input features.

The transform method is relatively straightforward to describe, and can be defined as such:

$$Transform(x, E) \coloneqq [\,\Theta(\mathrm{x}, E_1), \Theta(\mathrm{x}, E_2), \ldots, \Theta(\mathrm{x}, E_t)\,] \quad s.t.\ x \in \mathbb{R}^m, E \in (\mathbb{Z}_+)^{\mathrm{t\,X\,m}}$$

$$\Theta(x, \varepsilon) \coloneqq \prod_{i=1}^{m} x_i^{\varepsilon_i} \quad s.t.\ x \in \mathbb{R}^m, \varepsilon \in (\mathbb{Z}^+)^m$$

Where:

- **m** is the number of features in the data,
- **t** is the number of features in the transformed space
- **x** is a m-dimensioned vector representing a data point,
- **E** is a **t** by **m** matrix over the positive integers that holds the exponents used to map an input vector to the transform space (and will be created during the fitting process),
- **ε** is a **m**-dimensional vector for each feature in the transform vector containing the exponents for each term in the input vector.

The process for creating polynomial features from an input vector can be represented as a tree with a depth equal to the degree of the polynomial feature space, (maximum) branching factor equal to the dimension of the input vector, and the root node holding a value of 1. *(Property 1)*

The children of each node are equal to the product of the parent's value and the term from the input vector that it corresponds with (the leftmost child corresponds with the first term, the rightmost child corresponds with the last term, and the correspondence of all children in between can be inferred). This correspondence with terms from the input will also determine the relative ordering of the children, which determines the traversal order. *(Property 2)*

The values of each node can be read out in breadth-first order, skipping nodes with duplicate values (and their children) to yield the transformed input vector. *(Property 3)*

An example of this process is shown below (albeit poorly), performing degree 3 polynomial feature transformation on a 2-dimensional vector containing the values [a, b]:

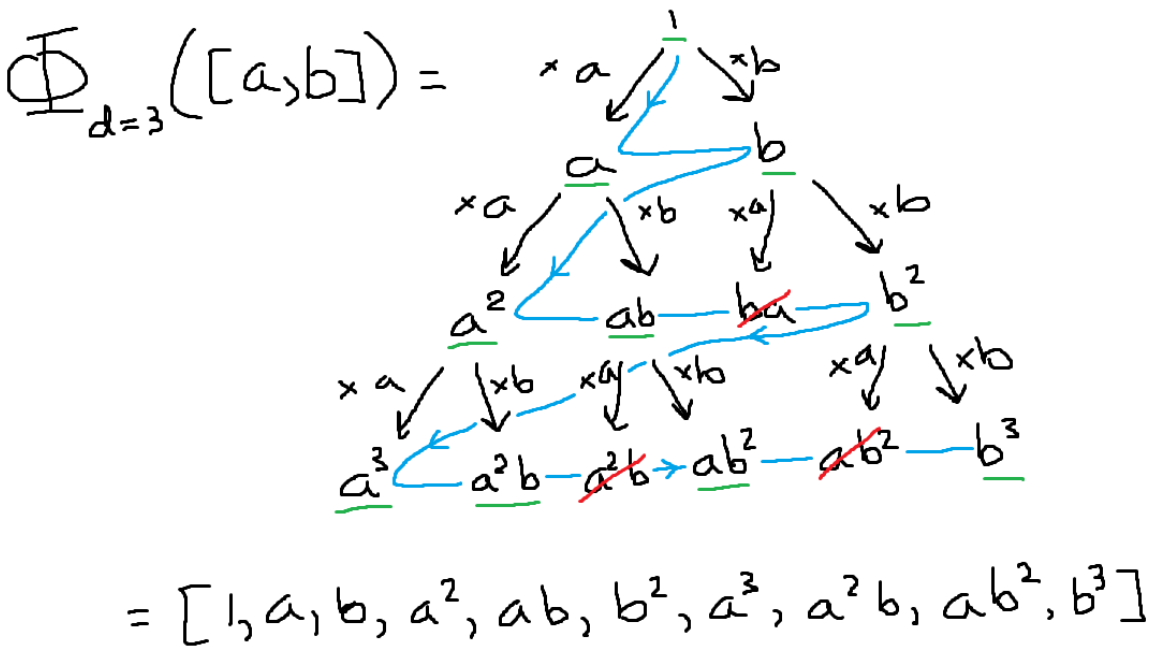

Figure 8.2.1

Because we're only keeping the exponents for each term as E, the process described in figure 8.2.1 will be used to obtain the exponents for each term in the transform space (e.g., the "a^3" term would be represented by [3, 0], as $a^3 \equiv a^3b^0$). *(Property 4)*

From figure 8.2.1, we can also tell that all nodes of the same degree will be at the same depth; and because all duplicate nodes will necessarily have the same degree, we can see that all duplicate nodes will always be at the same depth.

Thus, the advantages of reading out the values of the tree in breadth-first order (prop. 1) following the child—input-term correspondence (prop. 2) are 3-fold:

- First, it's easy to describe.
- Second, which nodes to prune can be determined from the structure of the tree, where incrementing the exponent of a term lower in sequence (prop. 2) than any term already incremented in the current value of the node (i.e., traversing a branch "more left" than any branch travelled in the path from the root) will result in a duplicate[†]. *(Property 2.5)*

  For example, in figure 8.2.1 we can see the term $\mathbf{a^2b}$ in the final level get pruned when it is found by increasing the degree from the term **ab** (since we already found this term from $\mathbf{a^2}$). We can see the path from the root to this node already makes a "right" turn when moving from term **a** at degree 1 to term **ab** at degree 2, so any "left" turns will result in a duplicate term (when reading out values according to prop. 2).
- Third, and perhaps most importantly, the resulting transformed vector will have its terms in the same order as the vector created by scikit-learn's implementation of polynomial features, which makes testing a lot easier for me[‡].

From the properties established (1-4) and the subsequent property (2.5) described in the second point above, an algorithm can be (and has been) designed to compose and traverse this tree while storing m-dimensional vectors of exponents for each term in the transform space to be used by the transform function described at the beginning of section 8.2.

Since the exponent vectors are all that's needed for transformation, the tree itself doesn't need to be kept after fitting; so the algorithm should be (and has been) designed to compose and traverse the tree in tandem when fitting while only storing the information necessary for traversal (i.e., the parent-child relationships are "forgotten" after a level of the tree has been fully traversed, so only the values of nodes are kept).

[†] The proof on why this fact holds will be left as an exercise for the reader 😊.

[‡] While the order of the resulting transformed vector wouldn't have any effect on the results of fitting the model, it certainly saves me a lot of headache when comparing my results with tools that I'm trying to replicate (but I suppose the resulting weights vector would be out of order which is undesirable); and I find staying consistent with existing libraries to be aesthetically pleasing.

The algorithm used to obtain exponents in this way is as follows:

Algorithm 1: Obtaining exponents for polynomial feature transformation

```
Input  : m ∈ ℤ₊, the dimension of the input vector;
         d ∈ ℤ₊, the degree of the polynomial transformation;
Output : E ∈ (ℤ₊)^(t×m), a matrix of shape (t, m) holding exponent values

1.  E ← [[0_m]]                          # create root with zero vector
2.  for depth ∈ [1,d] do
3.  |   level ← []                       # holds children at current depth
4.  |   for parent ∈ E_-1 do             # for each node in previous depth
5.  |   |   last ← index of last non-zero value in parent,
    |   |             0 if parent is 0_m
6.  |   |   for i ∈ [last,m] do
7.  |   |   |   child ← copy parent
8.  |   |   |   increment child_i
9.  |   |   |   append child to level
10. |   append level to E
11. return flattened E                   # concatenate all lists in E
```

From algorithm 1 it is (somewhat) apparent that all properties stated are upheld. The root node is established at line 1 containing an m-dimensional zero vector (which, when applied to any m-dimensional input vector using the Θ formula described at the beginning of section 8.2, will always yield 1).

On line 2, we iterate for each level of the tree, but we never use the iterator as knowing the current depth of traversal isn't necessary (since we only ever use the most recent depth traversed to create the next depth of the tree, as seen in line 4).

Line 5 specifically is what upholds property 2.5, where we find the index of the last incremented term (which encodes the last "turn" made to reach the parent node) and use that as a lower bound for what child nodes to expand on lines (6-9).

Line 11 flattens E from a list of lists of exponent vectors to a list of exponent vectors by concatenating each element in E into one large list of exponent vectors.

Below is a brief result of comparing this algorithm to scikit-learn's implementation (which can also be found in '**src/addendum.ipynb**'):

```
>>> from sklearn.preprocessing import PolynomialFeatures as SKPF
>>> d = 4; x = np.array([[1,2,3,4,5,6]])
>>> all((SKPF(degree=d).fit_transform(x) == PolynomialFeatures(degree=d).fit_transform(x))[0])
True
```