Laptop Price Prediction

EE 559 Course Project

Data Set: Laptop Prices

Anuja Shah, <apshah@usc.edu>
Shounak Das, <shounakd@usc.edu>

April 15, 2024

1 Abstract

Given a dataset with specific characteristics of numerous laptops, we attempted to predict the price of laptops provided in novel and unseen data. Text language from our training data was cleaned up, translated, and encoded into data that our ML models could interpret. Some features were added, combined, or changed from the original data, so as to provide more information from which the models could accurately predict prices. We used 1-Nearest Neighbor and Linear Regression models for our baselines, as well as a trivial system that averaged the price of all the laptops. We tested the performance of k-NN, MLP, RBF Network, Decision Tree Regression, Ridge Regression, and SVR. Performance was measured using three metrics: RMSE, MAE, and R2-Score. Our best performing model was the Ridge Regression model, with a regularization parameter of alpha = 1. The three performance metrics measuring the distance (error) between the true and predicted prices were most greatly minimized with this model. The simplicity of the Ridge Regression model worked well on our dataset, which was already small and simplistic itself. Our trained Ridge Regression mitigated overfitting and outperformed the baseline models and trivial system when used on the test dataset. Thus, our trained Ridge Regression model would be able to accurately predict prices of novel laptop data if given similar characteristics to work with as those found in our training set.

2 Introduction

2.1 Problem Assessment and Goal

We chose the laptop dataset, which included data for the Company, TypeName, Inches, ScreenResolution, Cpu, Ram, Memory, Gpu, OpSys, Weight, and Price. Using this data set to train models, we want to be able to

accurately predict the price of a laptop for novel data. We want to compare the performance of several models, such as k-Nearest Neighbors (kNN), Linear Regression, Multilayer Perceptron (MLP), Radial Basis Function (RBF) Network, Decision Tree Regression, Ridge Regression, Support Vector Regression (SVR), and a Trivial System. Once comparisons are complete, our goal is to determine the most effective model for laptop price prediction.

3 Approach and Implementation

3.1 Dataset Usage

The training dataset contains 901 data points, while the test dataset contains 402 data points. Pre-processing allowed us to determine which categories were able to be split into more, and how they should be encoded. For some categories, they were entirely numerical but included strings for weight or GBs, which had to be removed so the data could be treated as floats or integers. Originally, the datasets included 11 categories, but after pre-processing this was expanded to 15 categories, and finally, 46 categories after feature engineering and encoding. The pre-processing and encoding methods had to be able to be duplicated on the test dataset, which meant that we also had to ensure that unknown/unseen data could be dealt with in our encoding scheme.

Once the pre-processing and feature engineering (the most time-consuming steps of our project) was completed, we were able to finally move on to feature learning. We used a k-fold split (with k = 5) to split the training set as 900/5 = 180 data points in the validation set. Thus, we used an 80/20 split and tested the performance of the model, using the results to fine tune parameters and ensure the model was not underfit or overfit. Finally, we used our test set to get a conclusive and representative measure of performance for each baseline model and the final chosen system with the best performance.

3.2 Preprocessing

Step 1: Importing Training Dataset and Splitting Features

- We imported the data from ‘laptop_data_train.csv’ and split some features into multiple features.
- *Note*: By making everything lowercase and removing spaces, we were able to reduce the number of unique values in features for repetitive data that may have had only a slight difference in capitalization or spacing

Step 2: EncodingWe determined how to encode the categorical data to numerical data. Our choices were one-hot encoding and label encoding. *Note*: For the Touchscreen feature, we used binary encoding, where 1 meant that Touchscreen was mentioned in the description, and 0 meant it was not.

Option 1 — One-Hot Encoding

- *Pros*: Preserves uniqueness (each category gets its own binary column, i.e. no ordinal relationship imposed), and it works well with nominal categorical data where there is no intrinsic order.
- *Cons*: Increases dimensionality, and adds sparse matrices which can consume more memory and computational resources.
- Chosen features: Company, TypeName, and OpSys

Option 2 — Label Encoding

- *Pros*: Reduced dimensionality (one column for labels) saves memory and computational resources, preserves order, and simplicity (straightforward to implement)
- *Cons*: May inadvertently introduce ordinal relationships where none exist, thus leading to potentially biased models
- Chosen features: Display, MemoryComponent1, MemoryComponent2, CpuModel, and Gpu
    - If MemoryComponent2 was empty, it was assigned as “None”, which was then included in the label encoding

The preference of one-hot encoding for nominal categorical data with no inherent order vs. the preference for label encoding for ordinal categorical data with a clear order led us to decide which encoding to use based on the characteristics of each feature. Below, we have an example of how we chose to label encode the MemoryComponent1 based on the average pricing of each feature (a similar approach was used for the other features):

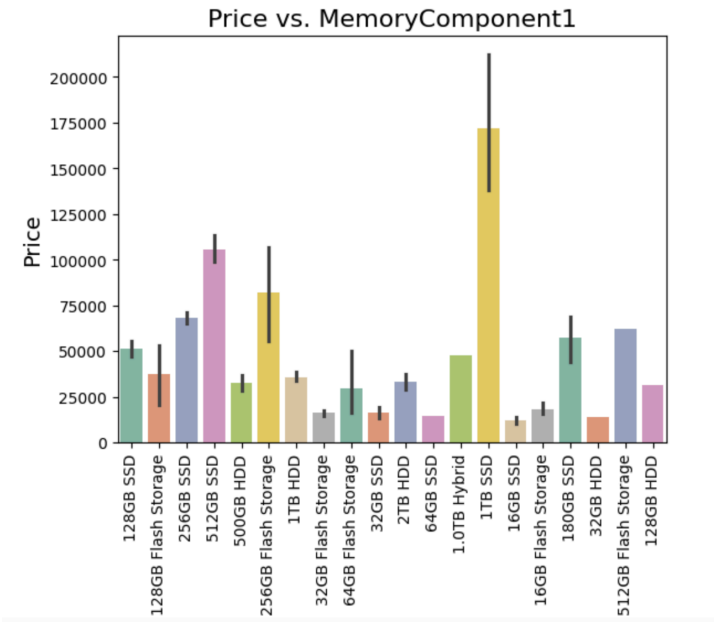

Fig. 1 Price vs. Memory Component 1 Storage Type

- Rank of these storage types in (ascending) order of how much they affect the price of a laptop, considering price:

MemoryComponent1	
16GB SSD	12360.960000
32GB HDD	14119.200000
64GB SSD	14811.307200
32GB Flash Storage	16122.626667
32GB SSD	16356.960000
16GB Flash Storage	18476.438400
64GB Flash Storage	29646.708800
128GB HDD	31435.200000
500GB HDD	32560.845830
2TB HDD	33339.503314
1TB HDD	35962.712676
128GB Flash Storage	37721.440800
1.0TB Hybrid	47898.720000
128GB SSD	51401.596686
180GB SSD	57196.080000
512GB Flash Storage	62071.200000
256GB SSD	68135.122432
256GB Flash Storage	82206.644400
512GB SSD	105603.977388
1TB SSD	171957.648000

Fig. 2 MemoryComponent1 storage type rankings according to mean price

- *Note*: For the label encoded features, if there was a feature in the test data set that did not exist in the training data set, it was assigned a value of "-1"

Step 3: Normalization

The rest of the categories were numerical and did not require encoding: Inches, Width, Height, ClockSpeed, Ram, Weight, and Price. All that was required was removing unnecessary text, like units, included with the numerical data. The data was all stored as strings, so conversion to integers and floating point numbers was necessary. Finally, normalization was applied to all categories that weren't encoded and the label encoded categories, enforcing consistency in scale and adding a regularizing effect by avoiding large parameter updates, thus preventing overfitting and improving the generalization ability of the models.

3.3 Feature engineering

We developed a few new features by splitting existing features into multiple features, allowing for more specificity:

Features Before: (Total = 11)	Features After Preprocessing: (Total = 15)	Features After Encoding: (Total = 46)
Company: 19 TypeName: 6 Inches: 17 ScreenResolution: 37 Cpu: 100 Ram: 8 Memory: 33 Gpu: 96 OpSys: 9 Weight: 161 Price: 600	Company: 19 TypeName: 6 Inches: 17 Display: 10 Touchscreen: 2 ScreenArea: 15 CpuModel: 81 ClockSpeed: 25 Ram: 8 MemoryComponent1: 20 MemoryComponent2: 6 Gpu:90 OpSys:9 Weight:157 Price:600	one_hot_Company_Acer: 2 one_hot_Company_Apple: 2 one_hot_Company_Asus: 2 one_hot_Company_Chuwi: 2 one_hot_Company_Dell: 2 one_hot_Company_Fujitsu: 2 one_hot_Company_Google: 2 one_hot_Company_HP: 2 one_hot_Company_Huawei: 2 one_hot_Company_LG: 2 one_hot_Company_Lenovo: 2 one_hot_Company_MSI: 2 one_hot_Company_Mediacom: 2 one_hot_Company_Microsoft: 2 one_hot_Company_Razer: 2 one_hot_Company_Samsung: 2 one_hot_Company_Toshiba: 2 one_hot_Company_Vero: 2 one_hot_Company_Xiaomi: 2 one_hot_TypeName_2 in 1 Convertible: 2 one_hot_TypeName_Gaming: 2 one_hot_TypeName_Netbook: 2 one_hot_TypeName_Notebook: 2 one_hot_TypeName_Ultrabook: 2 one_hot_TypeName_Workstation: 2 Inches: 17 Display: 10 Touchscreen: 2 ScreenArea: 15 CpuModel: 81 ClockSpeed: 25 Ram: 8 MemoryComponent1: 20 MemoryComponent2: 6 Gpu: 90 one_hot_OpSys_Android: 2 one_hot_OpSys_Chrome OS: 2 one_hot_OpSys_Linux: 2 one_hot_OpSys_Mac OS X: 2 one_hot_OpSys_No OS: 2 one_hot_OpSys_Windows 10: 2 one_hot_OpSys_Windows 10 S: 2 one_hot_OpSys_Windows 7: 2 one_hot_OpSys_macOS: 2 Weight: 157 Price: 600

Fig. 3 Number of features and classes before, after preprocessing, and after encoding

- We split the ScreenResolution category into 4 features: Display (i.e IPS Panel, Full HD, etc.), Touchscreen (i.e. Yes or No), Width, and Height (width and height referring to pixel count).

    - Then, we developed a new feature called ScreenArea, which multiplied the Height and Width categories together into one column
- We split the CPU category into 2 features: CpuModel (i.e. Intel Core i7, Intel Core i5 7200U) and ClockSpeed (i.e. 2.7 GHz). We checked the CPU for extra spaces, and there were none.
- We split Memory into MemoryComponent1 + MemoryComponent2.
- Even though GPU has 90 unique feature values, we decided against processing it further (besides stripping extra spaces) because each feature of a single GPU is extremely important in pricing, so there is no need to split the feature further, since they need to be considered all together (i.e. model, series, brand, etc.).
- After encoding, feature space was expanded to 46

3.4 Feature dimensionality adjustment

The feature dimensionality of our dataset falls within the optimal range for training a model based on the criterion that Nc (No. of constraints) -> (3-10) d.o.f [degrees of freedom].

Given that our dataset has approximately 720 training samples (after the cross-validation split), the optimal number of categories that our dataset can have is limited to 72 based on the above criterion, as 720/10 = 72. This motivated our decision to apply label encoding on categories that had a high number of unique feature values like Gpu and CpuModel, for example. This helped us constrain the number of columns in the final dataset, which could have blown up with a different choice of encoding like one-hot encoding.

We have also opted to adjust the dimensionality of our data based on model performance after the feature engineering stage. During feature learning, it was observed that the predictions aren't reliable across different models. In order to mitigate this issue, we decided to reduce the complexity of our dataset by extracting the features that were most pertinent to the model's learning.

We achieved this by implementing two feature selection techniques: Sequential Forward Selection (SFS) and Univariate Feature Selection (UFS). SFS searches through the space of feature subsets, evaluating the performance of the model with each subset, and selecting the subset that maximizes the chosen performance metric, which is the minimization of the mean squared error in our case. It is extremely powerful for feature selection as it considers the impact of removing or adding features on the overall model performance. UFS ranks features based on statistical tests, which in our case was an F-value between each feature and the target variable in a regression task, but it does not consider feature interactions.

It can be beneficial to use one technique over the other in some cases due to how they affect model performance, as can be observed in 3.5.

3.5 Training, Classification or Regression, and Model Selection

Common to all models:

k-fold Cross Validation, k = 5: Splits the dataset into 5 folds with 4 training folds and 1 validation fold. The model is trained iteratively 5 times to compare performance over different selections of validation folds.

3.5.1 Linear Regression

In Linear Regression, the relationship between the input features (x) and the target variable (y) is represented by a discriminant function. The goal of training is to find the optimal values for the weights that minimize the difference between the predicted and actual target values.

Model Parameters:

- Tolerance and range adjustment, tolerance = 5000: Outliers were observed among model predictions when some outputs were in the order of 10e13. In order to combat these anomalies, we came up with an ad-hoc rule to substitute all outlier values with the mean value of all the price label values in the training dataset. A prediction is considered an anomaly when its value lies outside the tolerance range. The tolerance range is defined by the range of price label values +- 5000 on either side of the range.
- Learning Rate: This Linear Regression model does not have a learning rate parameter because the optimization is not based on iterative optimization algorithms like gradient descent. Instead, it directly computes the optimal coefficients using the closed-form solution provided by the fit() method.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	15738.0719	498.7242
MAE	11495.8372	290.3778
R-squared Error	0.8263	0.0128

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	18485.3826	1507.6078
MAE	13075.3512	974.1267
R-squared Error	0.7523	0.0581

The baseline Linear Regression model performs best with the UFS feature selection as it provides the most balanced dataset with standardized values. The model's performance was consistently good across all tested methods but using UFS gave us the best generalized model as the training and validation performances were the most similar.

3.5.2 1-Nearest Neighbour

The 1-Nearest Neighbor (1-NN) model is a lazy learning algorithm used for classification. It memorizes the entire training dataset and assigns labels based on the closest data point. It calculates distances, typically using the Euclidean distance, between new and training data points to identify the nearest neighbor, assigning its label to the new data point.

Algorithm Overview:

For each new data point in the test set:

- Calculate the distance between the new data point and every data point in the training set.
- Identify the nearest neighbor based on the calculated distances.
- Assign the label of the nearest neighbor to the new data point as its predicted label.

Model Parameters:

- k-fold Cross Validation, k = 5: Splits the dataset into 5 folds with 4 training folds and 1 validation fold. The model is trained iteratively 5 times to compare performance over different selections of validation folds.
- n_neighbors: This parameter defines the number of nearest neighbors to consider when making predictions. In this case, it's set to 1, indicating that only the single closest neighbor is considered for prediction.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	1485.6598	161.0664
MAE	210.5236	26.8479
R-squared Error	0.9984	0.0003

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	16227.7738	2464.3353
MAE	10686.4752	1235.1701
R-squared Error	0.8064	0.0704

The 1-Nearest Neighbor Model is very prone to overfitting as its predictions are based on the closest data point, and therefore feature selection is necessary. The model trained to produce its best results using the SFS technique as

shown above. In this case, the model is still slightly overfit but offers some kind of generalization when compared to the other iterations.

3.5.3 Ridge Regression

Ridge Regression is a linear regression technique that extends ordinary least squares (OLS) regression by adding a penalty term to the loss function. This penalty term, also known as L2 regularization, helps to prevent overfitting by imposing constraints on the magnitude of the coefficients.

Algorithm Overview:

Ridge Regression minimizes the following Mean Squared Error (MSE) objective function:

$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{j=1}^{P} x_{i,j}\theta_j\right) - y_i\right]^2 + \gamma\frac{1}{2}\sum_{j=1}^{P}\theta_j^2$$

- ɣ is the regularization parameter (also known as the tuning parameter or penalty term).
- P is the number of features.
- θ j represents the coefficients of the model.

Regularization: The additional term in the objective function penalizes large coefficients, effectively shrinking them towards zero. This helps to reduce the complexity of the model and prevent overfitting.

Optimization: The Ridge Regression model is trained using optimization techniques such as gradient descent or closed-form solutions to minimize the objective function. The optimal coefficients are found by balancing the trade-off between minimizing the MSE and minimizing the regularization term.

Model Parameters:

- Regularization Strength (α): The alpha parameter in the code (ɣ in the above equation) specifies the regularization strength or penalty term in the Ridge Regression model. It controls the amount of shrinkage applied to the coefficients to prevent overfitting. In ourcode, it's initialized with a value of 1.0.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	15783.0635	491.1236
MAE	11533.9280	284.9711
R-squared Error	0.8254	0.0126

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	16700.4669	2068.0743
MAE	12105.7210	1043.7598
R-squared Error	0.7978	0.0515

The Ridge Regression model performs consistently across the default dataset, standardized dataset, SFS selection, and UFS selection runs. However, the validation results are slightly better in the UFS method and therefor it is selected as the best model for our problem statement. It also helps that the UFS model uses the least number of features (28) to make its predictions when compared to SFS (37) and the default dataset (46) in having the best generalization.

3.5.4 Multi-Layer Perceptron (MLP):

The Multi-Layer Perceptron (MLP) Regression model is a type of artificial neural network (ANN) used for regression tasks. It consists of multiple layers of interconnected neurons, including input, hidden, and output layers. Each neuron in the network performs a weighted sum of its inputs, applies an activation function, and passes the result to the next layer. Because of the small size of the dataset, we chose three hidden layers using ReLU activations for our MLP model.

Algorithm Overview:

Forward Propagation: Propagate the input data through the network to obtain predictions. This involves computing the output of each neuron in each layer based on the weighted sum of its inputs and applying an activation function.

Loss Calculation: Compute the loss (error) between the predicted output and the actual target values using a suitable loss function, such as Mean Squared Error (MSE) or Mean Absolute Error (MAE).

Backpropagation: Use an optimization algorithm, such as gradient descent or its variants, to minimize the loss function by updating the weights and biases of the network. This is done by iteratively adjusting the parameters in the opposite direction of the gradient of the loss function with respect to the parameters.

Training: Iterate through the dataset multiple times (epochs), performing forward propagation, loss calculation, and backpropagation to update the parameters and improve the model's performance.

Model Parameters:

- Hidden Layer Sizes: The hidden_layer_sizes parameter specifies the number of neurons in each hidden layer of the MLP model. In this case, it's set to (128, 256, 128), indicating that there are three hidden layers with 128 neurons in the first hidden layer, 256 neurons in the second hidden layer, and 128 neurons in the third and final hidden layer.
- Activation Function: The activation parameter specifies the activation function used by the neurons in the MLP model. In this case, it's set to 'relu', which stands for Rectified Linear Unit. This activation function is commonly used in deep learning models for its ability to introduce non-linearity and handle vanishing gradient problems.
- Learning Rate: Adam Optimizer with learning rate of 0.005.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	15307.6253	358.9466
MAE	10654.4254	144.1472
R-squared Error	0.8358	0.0088

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	15870.5257	1237.9300
MAE	11077.7631	540.9145
R-squared Error	0.8200	0.0280

The model performed much better on standardized data than the default dataset as this brought all the feature values to scale. This allowed for a more localized iterative learning process that is more efficient for an MLP. The best results (displayed above) are obtained after the UFS feature selection technique is applied as this provides the most balanced dataset with the lowest feature complexity (28). Training on fewer features is also good practice for better model generalization.

3.5.5 k-Nearest Neighbors Regression

The k-Nearest Neighbors (k-NN) Regression model is a non-parametric, instance-based learning algorithm used for regression tasks. In k-NN regression, predictions for a new data point are made by averaging the target values of its k nearest neighbors in the training set.

Algorithm Overview:

Initialization: There is no explicit training phase in k-NN regression. The model simply stores the training dataset.

Prediction: For each new data point in the test set: Calculate the distance between the new data point and every data point in the training set. Identify the k nearest neighbors based on the calculated distances. Predict the target value by averaging the target values of the k nearest neighbors.

Model Parameters:

- n_neighbors: This parameter defines the number of nearest neighbors to consider when making predictions. In this case, it's set to 5, indicating that only the 5 closest neighbors are considered for prediction.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	12672.4222	399.9753
MAE	8117.6892	182.2988
R-squared Error	0.8873	0.0091

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	15340.5478	1540.8463
MAE	9862.3397	541.3509
R-squared Error	0.8332	0.0224

The k-NN model performs very well across all training runs, so the best model selection is based on which training technique has the best generalization. As good generalization practice, the dataset is standardized. Between the feature selection techniques, we select the SFS model as it performs best across the validation datasets.

3.5.6 Support Vector Regression

Support Vector Regression (SVR) is a regression algorithm based on Support Vector Machines (SVMs). It works by finding the hyperplane that best fits the data, while also maximizing the margin, which represents the distance between the hyperplane and the nearest data points (support vectors). SVR can capture both linear and non-linear relationships in the data by using different kernel functions.

Algorithm Overview:

Initialization: SVR begins by selecting a suitable kernel function, such as linear, polynomial, or radial basis function (RBF), and setting appropriate hyperparameters.

Feature Transformation: If necessary, SVR maps the input features into a higher-dimensional space using the selected kernel function. This transformation allows SVR to find non-linear decision boundaries in the original feature space.

Model Training: SVR identifies the hyperplane (decision boundary) that best separates the data points while maximizing the margin. In regression tasks, SVR aims to fit as many data points within a specified margin of tolerance (ε) around the hyperplane.

Optimization: SVR minimizes the loss function, which includes a term for the margin violation and a regularization term to prevent overfitting. This optimization process is typically performed using techniques such as gradient descent or quadratic programming.

Prediction: Once trained, SVR can make predictions for new data points by evaluating the learned hyperplane. The predicted values are obtained by taking the output of the hyperplane as the regression function.

Model Parameters:

- Kernel Type (kernel): The kernel parameter specifies the type of kernel function used in SVR. In this case, it's set to 'linear', indicating that a linear kernel is used.

```
Best Kernel Parameters:
{'C': 10, 'gamma': 0.01}
```

Model Performance:

```
Validation Set:
```

Error Metrics

Metric	Mean	Std Dev
RMSE	35052.8794	3373.2091
MAE	23368.5324	1105.9141
R-squared Error	0.1358	0.0433

The SVR model is observed to perform poorly on all models. This could be due to the size of the dataset, with the model unable to learn effectively due to the scarcity of data points within the margin. It also highlights that Support Vector Machines are better suited for classification tasks.

3.5.7 Radial Basis Function Network

Radial Basis Function (RBF) Regression utilizes kernel-based regression, where a kernel function measures similarity between data points in a high-dimensional feature space. It employs a single hidden layer of radial basis functions, centered at training data points, to compute activations based on similarity to input features. This approach predicts target values by weighting contributions of nearby data points.

Algorithm Overview:

Initialization: RBF Regression begins by selecting an appropriate kernel function, such as the Gaussian (RBF) kernel.

Feature Transformation: If necessary, RBF Regression maps the input features into a high-dimensional feature space using the selected kernel function. This transformation allows RBF Regression to capture non-linear relationships between input features and target variables.

Model Training: RBF Regression learns the model parameters by fitting a linear combination of radial basis functions to the training data. The model parameters include the coefficients of the basis functions and possibly other hyperparameters such as the width of the Gaussian kernel.

Optimization: RBF Regression minimizes a loss function, which typically includes a regularization term to prevent overfitting. The optimization process may involve techniques such as gradient descent or closed-form solutions, depending on the specific implementation.

Prediction: Once trained, RBF Regression can make predictions for new data points by evaluating the learned model on the input features. The predicted target values are obtained by combining the contributions of nearby data points weighted by their similarity to the input features.

Model Parameters:

- RBFSampler Parameters:

  gamma: The scaling parameter of the RBF kernel. It controls the width of the radial basis functions.

  n_components: The number of radial basis functions (features) to generate. It determines the dimensionality of the transformed feature space.

- Ridge Regression Parameters:

  alpha: The regularization parameter. It controls the amount of regularization applied to the regression model to prevent overfitting.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	11849.8547	373.7992
MAE	8782.7417	210.1800
R-squared Error	0.9015	0.0070

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	14785.7540	1305.8829
MAE	10521.1504	696.8339
R-squared Error	0.8426	0.0362

The RBF Regression Network performs really well on standardized data in comparison to non-standardized data. The best validation performance is recorded on SFS feature selected data. When feature dimensionality is reduced, the RBF network can map non-linear features in the lower dimension data more efficiently.

3.5.8 Decision Tree Regression

Decision Tree Regression is a non-parametric supervised learning algorithm used for regression tasks. It works by partitioning the feature space into regions, each associated with a predicted target value. The algorithm recursively splits the data based on feature thresholds to minimize the variance of the target variable within each partition.

Algorithm Overview:

Tree Construction: Decision Tree Regression constructs a binary tree structure, where each internal node represents a feature and a splitting criterion, and each leaf node represents a predicted target value.

Node Splitting: At each node, the algorithm selects the feature and threshold that maximize the reduction in variance of the target variable. This splitting criterion is typically determined using metrics like mean squared error (MSE) or variance.

Tree Pruning (Optional): Optionally, the tree may undergo pruning to prevent overfitting. Pruning involves removing nodes that do not significantly improve the model's performance on validation data.

Prediction: To make predictions for new data points, Decision Tree Regression traverses the tree from the root to a leaf node, following the splits based on the input features. The predicted target value for the new data point is the average (or another aggregation) of the target values in the leaf node.

Model Parameters:

- random_state: This parameter is used to control the randomness in the model. It sets the seed for random number generation, ensuring reproducibility of the results.
- Other parameters of the DecisionTreeRegressor, such as the maximum depth of the tree, minimum samples per leaf, and criterion for splitting, are left at their default values.

Model Performance:

Training Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	1048.1772	106.0105
MAE	211.6129	32.2145
R-squared Error	0.9992	0.0001

Validation Set:

Error Metrics

Metric	Mean	Std Dev
RMSE	17436.9156	958.0301
MAE	11350.4474	677.1792
R-squared Error	0.7802	0.0487

The Decision Tree regression model seems to be overfitting on all models due to the small size of the dataset. The UFS model is selected as the best model as it offers the best validation performance, and hence slightly better generalization. More data must be gathered for this model to perform better.

4 Results and Analysis: Comparison and Interpretation

Training Results

Baseline Models

Trivial System: Performed on a split data set with 0.85 training data and 0.15 validation data.

```
Training-Validation Split: 85.0%
```

```
Trivial Model's Laptop Price Prediction: 59077.37
```

```
Performance on Training Set:
```

RMSE	MAE	R-squared Error
37689.84424758283	28601.35737254902	0.0

```
Performance on Validation Set:
```

RMSE	MAE	R-squared Error
38491.53773183566	28367.38573529412	0.0

Linear Regression: Performed on completely feature engineered data, using 5-fold cross-validation and Univariate Feature Selection. Results stated in section 3.5.1.

1-Nearest Neigbor: Performed on completely feature engineered data, using 5-fold cross-validation and Sequential Forward Selection. Results stated in section 3.5.2.

Models' Performance Summary on Validation Dataset across 5-folds cross validation

Model	RMSE	MAE	R2-Score	Dataset Used

Ridge Regression	16700.4669	12105.721	0.7978	UFS
Multi-Layer Perceptron	15870.5257	11077.7631	0.82	UFS
Support Vector Regression	35052.8794	23368.5324	0.1358	UFS
k-Nearest Neighbors	15340.5478	9862.3397	0.8332	SFS

Additional Models' Performance Summary on Validation Dataset across 5-folds cross validation

Model	**RMSE**	**MAE**	**R2-Score**	**Dataset Used**
RBF Neural Network	14785.754	10521.1504	0.8426	SFS
Decision Tree Regression	17436.9156	11350.4474	0.7802	UFS

Testing Results

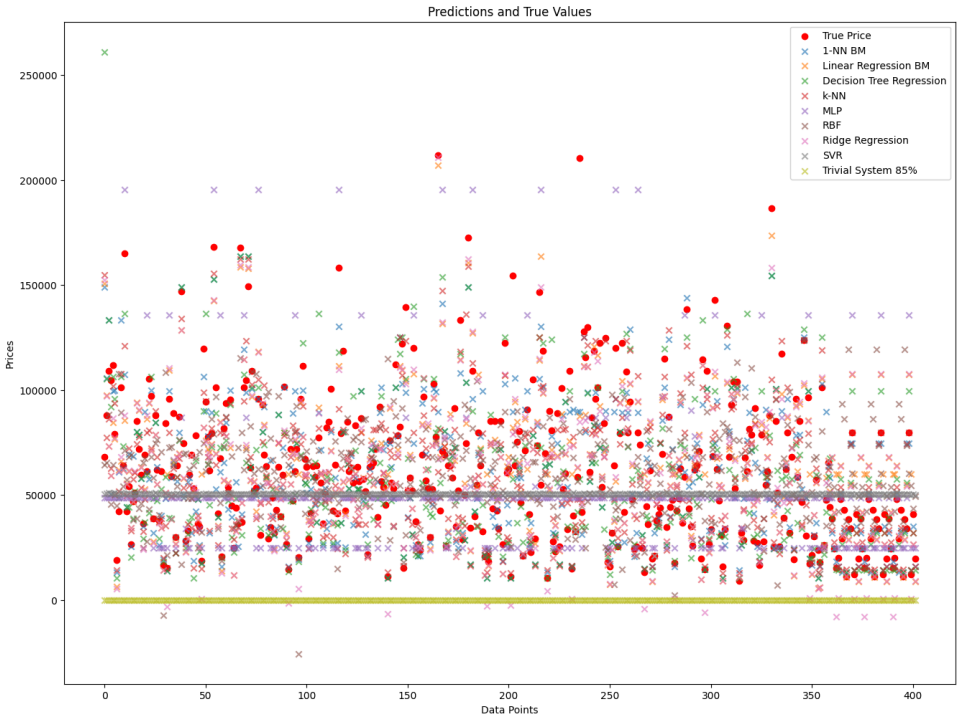

Fig. 4 Visualization of the price prediction by every model on the Test Dataset on each data point (instance).

Trivial System:

Trivial Model's Laptop Price Prediction: 59141.39

Performance on Testing Set:

RMSE	MAE	R-squared Error
35911.8597	27914.3228	0.0

1-Nearest Neighbor:

Error Metrics for Test Set:

Error Metrics

Metric	Value
RMSE	21271.2573
MAE	13492.2469
R-squared Error	0.6476

Linear Regression:

Performance on Testing Set:

Error Metrics

Metric	Value
RMSE	20627.2040
MAE	14982.0668
R-squared Error	0.6686

Clearly, the trivial system performs worse than both the required baseline systems, and linear regression is the best performing model. The baseline models use Feature Selection (SFS and UFS) on the test data predictions.

FINAL SYSTEM: Ridge Regression on Standardized UFS Dataset

```
Error Metrics for Test Set:
```

Error Metrics

Metric	Value
RMSE	19539.0609
MAE	14231.7027
R-squared Error	0.7027

5 Libraries used and what you coded yourself

Libraries used:

- numpy
- pandas
    - ‘get_dummies’ was used to one-hot encode
- sklearn
- matplotlib.py
- re
- rich
- seaborn
- tabulate

Coded ourselves:

- Complete feature preprocessing and feature engineering, including encoding, visualization plots, analysis and data characteristics.
- A Trivial System class that mimics an imported PyTorch NN module, and performs the desired operations to compute the output of the trivial system.
- Training Algorithm for all models, including cross validation and setting of hyperparameters.
- All visualization tables to report performance metrics of each model.
- Modularization of the code: Each stage of the project was split into different files for better accessibility of code, and functions were created to optimize the algorithm into shorter code blocks.
- Post-training and testing visualizations: Plots to provide a deep insight into each model’s predictions and performance.

6 Contributions of each team member

Anuja mainly worked on pre-processing, feature engineering, and the written report. Shounak mainly worked on feature engineering, feature learning, and the written report.

7 Summary and conclusions

Our approach focused on the preprocessing and feature engineering of the dataset. By ensuring our data was separated into appropriate categories and encoded effectively, we were able to ensure the majority of effective performance by the model. After that, minor tweaking to parameters for each model was all that was left to ensure the highest model performance. Results found the Ridge Regression model to be the best performing model as its simplicity complemented the dataset. It also mitigated overfitting which is helpful in performance. Notedly, our final model (Ridge Regression) performed better than the baseline models and trivial system on the test dataset.

As follow-on work, it would be useful to include more in-depth feature selection. While we coded functions for Principal Component Analysis (PCA) feature selection, we decided against implementing it due to time constraints. We used Univariate Feature Selection (UFS), and Sequential Forward Selection (SFS), but PCA transforms the original features into a set of orthogonal variables (principal components) which capture the maximum variance in the data. Feature selection is done by retaining those most important principal components which contain the maximum amount of variance. It would be interesting to observe whether this change in approach could improve the performance of the models that didn't do well in this experiment.

A key thing we learned was that preprocessing and feature engineering were far more time-consuming and important than we had previously considered. For our homeworks, our data was already clean and ready to use, so we did not understand how much time went into translating the data into something a model could interpret and work with before starting this project.

References

[1] P. Yadav, “Label encoding vs. One-hot encoding: Making sense of categorical data 👨‍💻,” Medium, https://medium.com/aimonks/label-encoding-vs-one-hot-encoding-making-sense-of-categorical-data-118191450 1f3 (accessed Apr. 10, 2024).

[2] “SVM with Univariate Feature Selection in Scikit learn,” GeeksforGeeks, https://www.geeksforgeeks.org/svm-with-univariate-feature-selection-in-scikit-learn/ (accessed Apr. 15, 2024).

[3] A. Harrag and T. Mohamadi, “‘PCA, SFS or LDA: What is the best choice for extracting speaker features,’” International Journal of Computer Applications, vol. 15, no. 3, pp. 1–3, Feb. 2011. doi:10.5120/1932-2578