Data Science III with python (Class notes)

STAT 303-3

Arvind Krishna, Emre Besler, and Lizhen Shi

3/24/23

Table of contents

Preface

These are class notes for the course STAT303-3. This is not the course text-book. You are required to read the relevant sections of the book as mentioned on the course website.

The course notes are currently being written, and will continue to being developed as the course progresses (just like the class notes last quarter). Please report any typos / mistakes / inconsistencies / issues with the class notes / class presentations in your comments here. Thank you!

Part I

Moving towards non-linearity

1 Introduction to scikit-learn

In this chapter, we'll learn some functions from the library `sklearn` that will be useful in:

1. Splitting the data into `train` and `test`
2. Scaling data
3. Fitting a model
4. Computing model performance metrics
5. Tuning model hyperparameters* to optimize the desired performance metric

**In machine learning, a model hyperparameter is a parameter that cannot be learned from training data and must be set before training the model. Hyperparameters control aspects of the model's behavior and can greatly impact its performance. For example, the regularization parameter λ, in linear regression is a hyperparameter. You need to specify it before fitting the model. On the other hand, the beta coefficients in linear regression are parameters, as you learn them while training the model, and don't need to specify their values beforehand.*

We'll use a classification problem to illustrate the functions. However, similar functions can be used for regression problems, i.e., prediction problems with a continuous response.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(font_scale=1.35)
```

Let us import the `sklearn` modules useful in developing statistical models.

```
# sklearn has 100s of models - grouped in sublibraries, such as linear_model
from sklearn.linear_model import LogisticRegression, LinearRegression

# sklearn has many tools for cleaning/processing data, also grouped in sublibraries
# splitting one dataset into train and test, computing cross validation score, cross valid
from sklearn.model_selection import train_test_split, cross_val_predict, cross_val_score
```

```
#sklearn module for scaling data
from sklearn.preprocessing import StandardScaler

#sklearn modules for computing the performance metrics
from sklearn.metrics import accuracy_score, mean_absolute_error, mean_squared_error, r2_sc
roc_curve, auc, precision_score, recall_score, confusion_matrix
```

```
#Reading data
data = pd.read_csv('./Datasets/diabetes.csv')
```

Scikit-learn doesn't support the formula-like syntax of specifying the response and the predictors as in the `statsmodels` library. We need to create separate objects for predictors and response, which should be *array-like*. A Pandas DataFrame / Series or a Numpy array are *array-like* objects.

Let us reference our predictors as object `X`, and the response as object `y`.

```
# Separating the predictors and response - THIS IS HOW ALL SKLEARN OBJECTS ACCEPT DATA (di
y = data.Outcome
X = data.drop("Outcome", axis = 1)
```

1.1 Splitting data into `train` and `test`

Let us create train and test datasets for developing a model to predict if a person has diabetes.

```
# Creating training and test data
    # 80-20 split, which is usual - 70-30 split is also fine, 90-10 is fine if the dataset
    # random_state to set a random seed for the splitting - reproducible results
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state =
```

Let us find the proportion of classes (*'having diabetes'* $(y = 1)$ *or 'not having diabetes'* $(y = 0)$) in the complete dataset.

```
#Proportion of 0s and 1s in the complete data
y.value_counts()/y.shape
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

Let us find the proportion of classes (*'having diabetes' ($y = 1$) or 'not having diabetes' ($y = 0$)*) in the train dataset.

```
#Proportion of 0s and 1s in train data
y_train.value_counts()/y_train.shape
```

```
0    0.644951
1    0.355049
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data
y_test.value_counts()/y_test.shape
```

```
0    0.675325
1    0.324675
Name: Outcome, dtype: float64
```

We observe that the proportion of 0s and 1s in the `train` and `test` dataset are slightly different from that in the complete `data`. In order for these datasets to be more representative of the population, they should have a proportion of 0s and 1s similar to that in the complete dataset. This is especially critical in case of imbalanced datasets, where one class is represented by a significantly smaller number of instances than the other(s).

When training a classification model on an imbalanced dataset, the model might not learn enough about the minority class, which can lead to poor generalization performance on new data. This happens because the model is biased towards the majority class, and it might even predict all instances as belonging to the majority class.

1.1.1 Stratified splitting

We will use the argument `stratify` to obtain a proportion of 0s and 1s in the `train` and `test` datasets that is similar to the proportion in the complete 'data.

```
#Stratified train-test split
X_train_stratified, X_test_stratified, y_train_stratified,\
y_test_stratified = train_test_split(X, y, test_size = 0.2, random_state = 45, stratify=y)
```

```
#Proportion of 0s and 1s in train data with stratified split
y_train_stratified.value_counts()/y_train.shape
```

```
0    0.651466
1    0.348534
Name: Outcome, dtype: float64
```

```
#Proportion of 0s and 1s in test data with stratified split
y_test_stratified.value_counts()/y_test.shape
```

```
0    0.649351
1    0.350649
Name: Outcome, dtype: float64
```

The proportion of the classes in the stratified split mimics the proportion in the complete dataset more closely.

By using stratified splitting, we ensure that both the `train` and `test` data sets have the same proportion of instances from each class, which means that the model will see enough instances from the minority class during training. This, in turn, helps the model learn to distinguish between the classes better, leading to better performance on new data.

Thus, stratified splitting helps to ensure that the model sees enough instances from each class during training, which can improve the model's ability to generalize to new data, particularly in cases where one class is underrepresented in the dataset.

Let us develop a logistic regression model for predicting if a person has diabetes.

1.2 Scaling data

In certain models, it may be important to scale data for various reasons. In a logistic regression model, scaling can help with model convergence. Scikit-learn uses a method known as gradient-descent *(not in scope of the syllabus of this course)* to obtain a solution. In case the predictors have different orders of magnitude, the algorithm may fail to converge. In such cases, it is useful to standardize the predictors so that all of them are at the same scale.

```
# With linear/logistic regression in scikit-learn, especially when the predictors have dif
# of magn., scaling is necessary. This is to enable the training algo. which we did not co
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test) # Do NOT refit the scaler with the test data, jus
```

1.3 Fitting a model

Let us fit a logistic regression model for predicting if a person has diabetes. Let us try fitting a model with the un-scaled data.

```
# Create a model object - not trained yet
logreg = LogisticRegression()

# Train the model
logreg.fit(X_train, y_train)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:763: Converge
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
LogisticRegression()
```

Note that the model with the un-scaled predictors fails to converge. Check out the data `X_train` to see that this may be probably due to the predictors have different orders of magnitude. For example, the predictor `DiabetesPedigreeFunction` has values in [0.078, 2.42], while the predictor `Insulin` has values in [0, 800].

Let us fit the model to the scaled data.

```
# Create a model - not trained yet
logreg = LogisticRegression()

# Train the model
logreg.fit(X_train_scaled, y_train)
```

```
LogisticRegression()
```

The model converges to a solution with the scaled data!

The coefficients of the model can be returned with the `coef_` attribute of the `LogisticRegression()` object. However, the output is not as well formatted as in the case of the `statsmodels` library since `sklearn` is developed primarily for the purpose of prediction, and not inference.

```
# Use coef_ to return the coefficients - only log reg inference you can do with sklearn
print(logreg.coef_)
```

```
[[ 0.32572891  1.20110566 -0.32046591  0.06849882 -0.21727131  0.72619528
   0.40088897  0.29698818]]
```

1.4 Computing performance metrics

1.4.1 Accuracy

Let us test the model prediction accuracy on the test data. We'll demonstrate two different functions that can be used to compute model accuracy - `accuracy_score()`, and `score()`.

The `accuracy_score()` function from the `metrics` module of the `sklearn` library is general, and can be used for any classification model. We'll use it along with the `predict()` method of the `LogisticRegression()` object, which returns the predicted class based on a threshold probability of 0.5.

```
# Get the predicted classes first
y_pred = logreg.predict(X_test_scaled)

# Use the predicted and true classes for accuracy
print(accuracy_score(y_pred, y_test)*100)
```

```
73.37662337662337
```

The `score()` method of the `LogisticRegression()` object can be used to compute the accuracy only for a logistic regression model. Note that for a `LinearRegression()` object, the `score()` method will return the model R-squared.

```
# Use .score with test predictors and response to get the accuracy
# Implements the same thing under the hood
print(logreg.score(X_test_scaled, y_test)*100)
```

```
73.37662337662337
```

1.4.2 ROC-AUC

The `roc_curve()` and `auc()` functions from the `metrics` module of the `sklearn` library can be used to compute the ROC-AUC, or the area under the ROC curve. Note that for computing ROC-AUC, we need the predicted probability, instead of the predicted class. Thus, we'll use the `predict_proba()` method of the `LogisticRegression()` object, which returns the predicted probability for the observation to belong to each of the classes, instead of using the `predict()` method, which returns the predicted class based on threshold probability of 0.5.

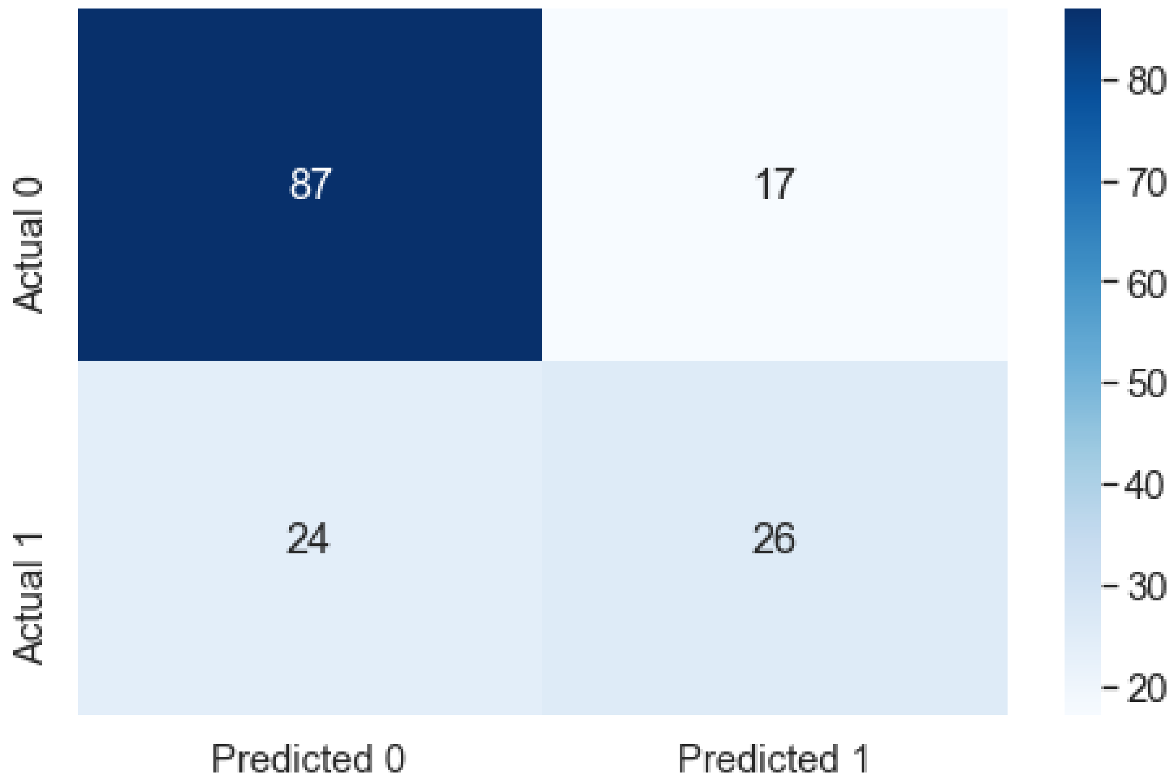
```
#Computing the predicted probability for the observation to belong to the positive class (
#The 2nd column in the output of predict_proba() consists of the probability of the observ
#belong to the positive class (y=1)
y_pred_prob = logreg.predict_proba(X_test_scaled)[:,1]

#Using the predicted probability computed above to find ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test, y_pred_prob)
print(auc(fpr, tpr))# AUC of ROC
```

```
0.7923076923076922
```

1.4.3 Confusion matrix & precision-recall

The `confusion_matrix()`, `precision_score()`, and `recall_score()` functions from the `metrics` module of the `sklearn` library can be used to compute the confusion matrix, precision, and recall respectively.

```
#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test, y_pred), columns=['Predicted 0', 'Predicted 1']
            index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```




```
print("Precision: ", precision_score(y_test, y_pred))
print("Recall: ", recall_score(y_test, y_pred))
```

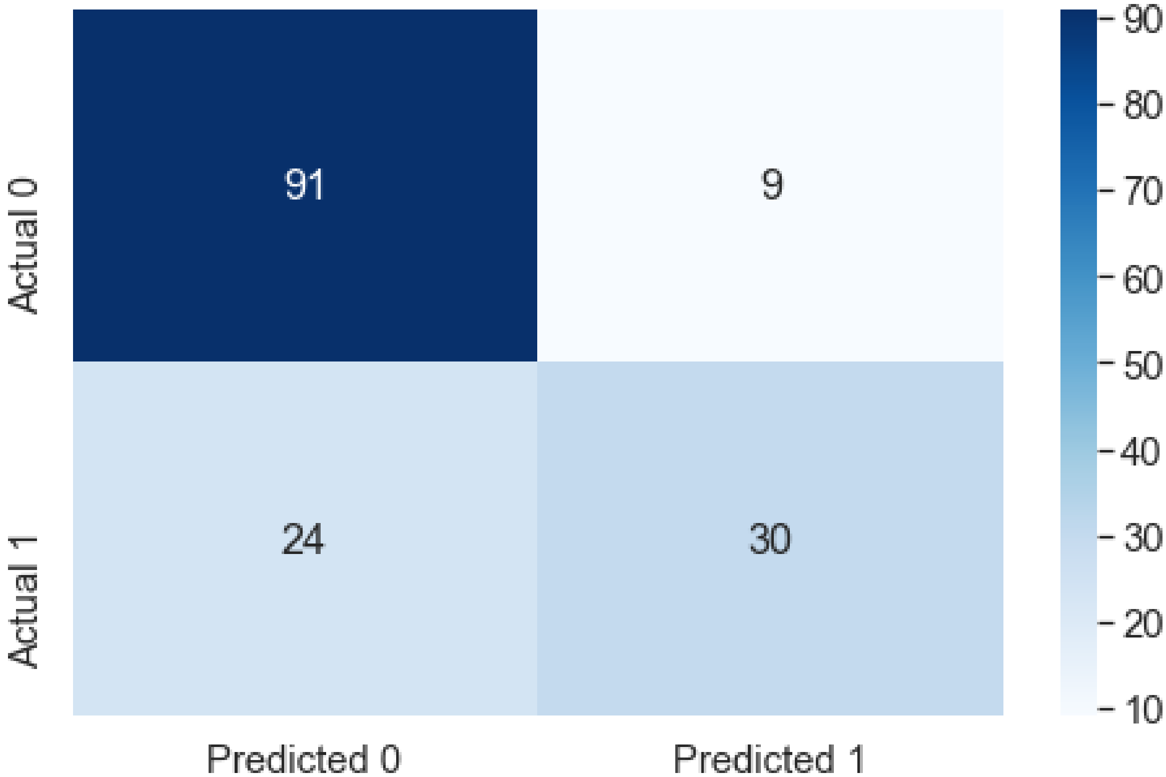
```
Precision:  0.6046511627906976
Recall:  0.52
```

Let us compute the performance metrics if we develop the model using stratified splitting.

```
# Developing the model with stratified splitting

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)

# Training the model
logreg.fit(X_train_stratified_scaled, y_train_stratified)
```

```
#Computing the accuracy
y_pred_stratified = logreg.predict(X_test_stratified_scaled)
print("Accuracy: ",accuracy_score(y_pred_stratified, y_test_stratified)*100)

#Computing the ROC-AUC
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(y_test_stratified, y_pred_stratified))
print("Recall: ", recall_score(y_test_stratified, y_pred_stratified))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_stratified), columns=['Predic
            index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  78.57142857142857
ROC-AUC:  0.8505555555555556
Precision:  0.7692307692307693
Recall:  0.5555555555555556
```



The model with the stratified train-test split has a better performance as compared to the other model on all the performance metrics!

1.5 Tuning the model hyperparameters

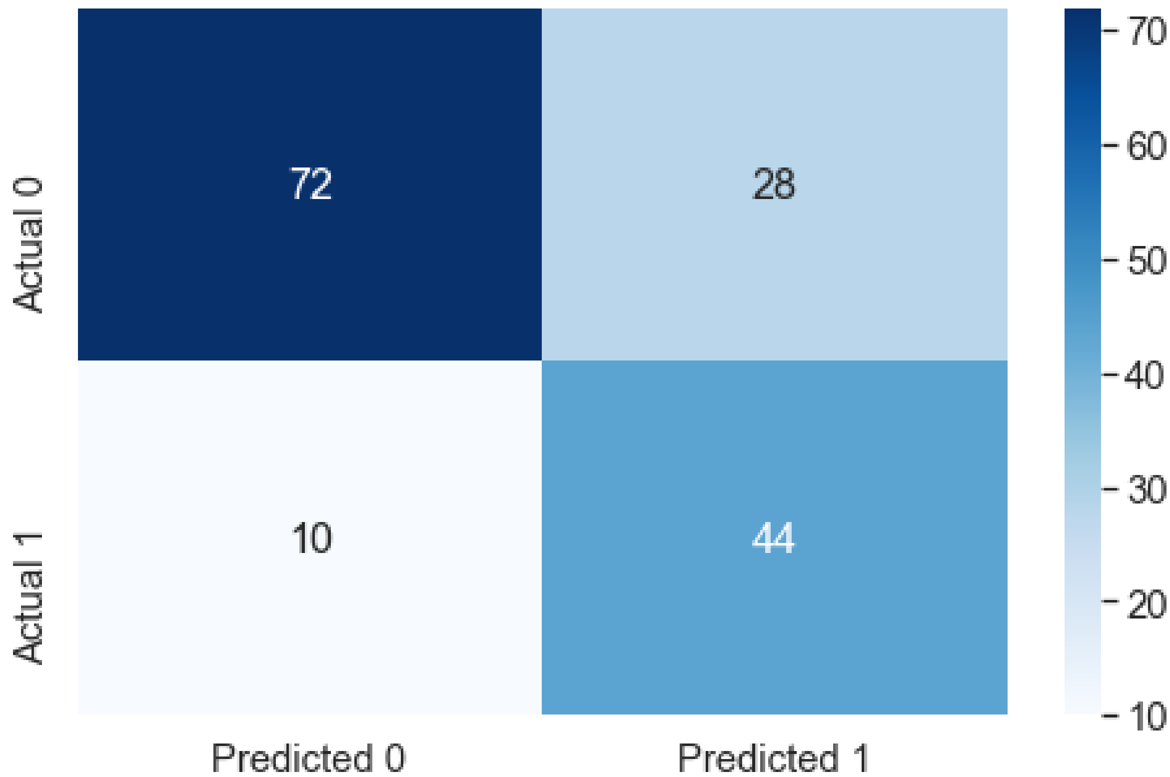
A hyperparameter (among others) that can be trained in a logistic regression model is the regularization parameter.

We may also wish to tune the decision threshold probability. Note that the decision threshold probability is not considered a hyperparameter of the model. Hyperparameters are model parameters that are set prior to training and cannot be directly adjusted by the model during training. Examples of hyperparameters in a logistic regression model include the regularization parameter, and the type of shrinkage penalty - lasso / ridge. These hyperparameters are typically optimized through a separate tuning process, such as cross-validation or grid search, before training the final model.

The performance metrics can be computed using a desired value of the threshold probability. Let us compute the performance metrics for a desired threshold probability of 0.3.

```python
# Performance metrics computation for a desired threshold probability of 0.3
desired_threshold = 0.3

# Classifying observations in the positive class (y = 1) if the predicted probability is g
# than the desired decision threshold probability
y_pred_desired_threshold = y_pred_stratified_prob > desired_threshold
y_pred_desired_threshold = y_pred_desired_threshold.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(y_test_stratified, y_pred_desired_threshold))
print("Recall: ", recall_score(y_test_stratified, y_pred_desired_threshold))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_desired_threshold),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1']
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  75.32467532467533
ROC-AUC:  0.8505555555555556
Precision:  0.6111111111111112
Recall:  0.8148148148148148
```



1.5.1 Tuning decision threshold probability

Suppose we wish to find the optimal decision threshold probability to maximize accuracy. Note that we cannot use the test dataset to optimize model hyperparameters, as that may lead to overfitting on the test data. We'll use K-fold cross validation on train data to find the optimal decision threshold probability.

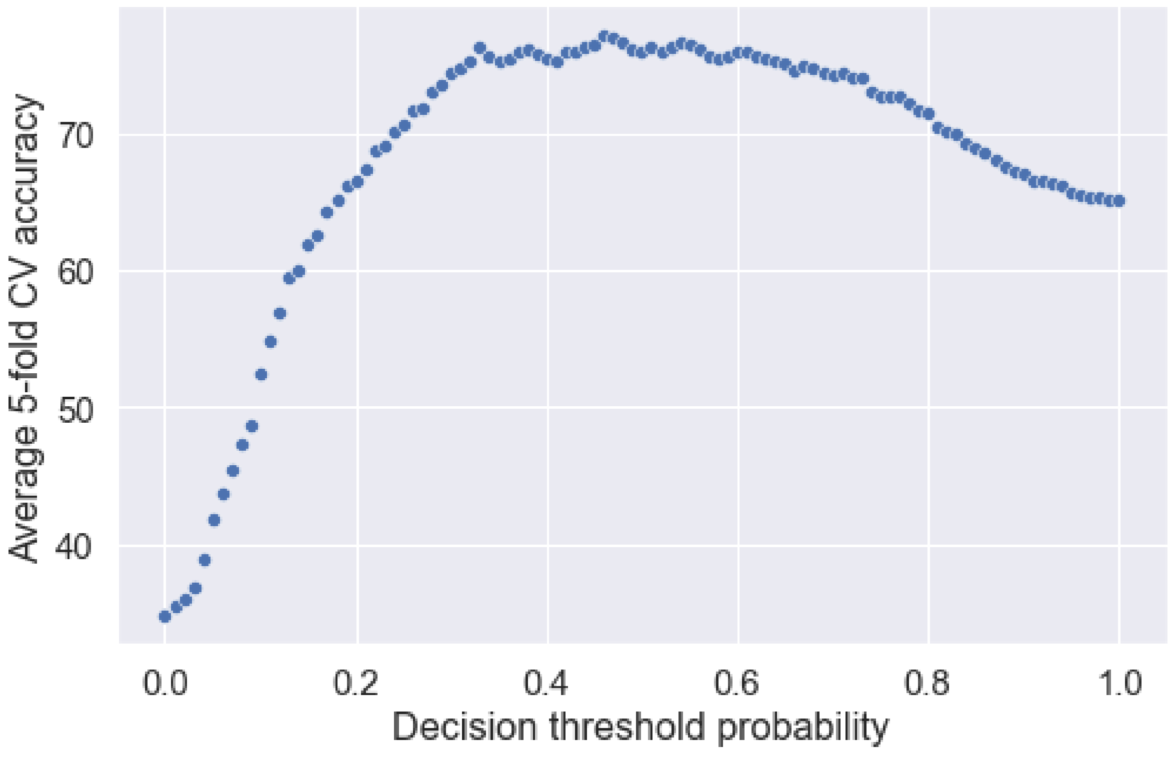
We'll use the `cross_val_predict()` function from the `model_selection` module of `sklearn` to compute the K-fold cross validated predicted probabilities. Note that this function simplifies the task of manually creating the K-folds, training the model K-times, and computing the predicted probabilities on each of the K-folds. Thereafter, the predicted probabilities will be used to find the one the optimal threshold probability that maximizes the classification accuracy.

```
hyperparam_vals = np.arange(0,1.01,0.01)
accuracy_iter = []

predicted_probability = cross_val_predict(LogisticRegression(), X_train_stratified_scaled,
```

```
                                              y_train_stratified, cv = 5, method = 'predic

for threshold_prob in hyperparam_vals:
    predicted_class = predicted_probability[:,1] > threshold_prob
    predicted_class = predicted_class.astype(int)

    #Computing the accuracy
    accuracy = accuracy_score(predicted_class, y_train_stratified)*100
    accuracy_iter.append(accuracy)
```

Let us visualize the accuracy with change in decision threshold probability.

```
# Accuracy vs decision threshold probability
sns.scatterplot(x = hyperparam_vals, y = accuracy_iter)
plt.xlabel('Decision threshold probability')
plt.ylabel('Average 5-fold CV accuracy');
```



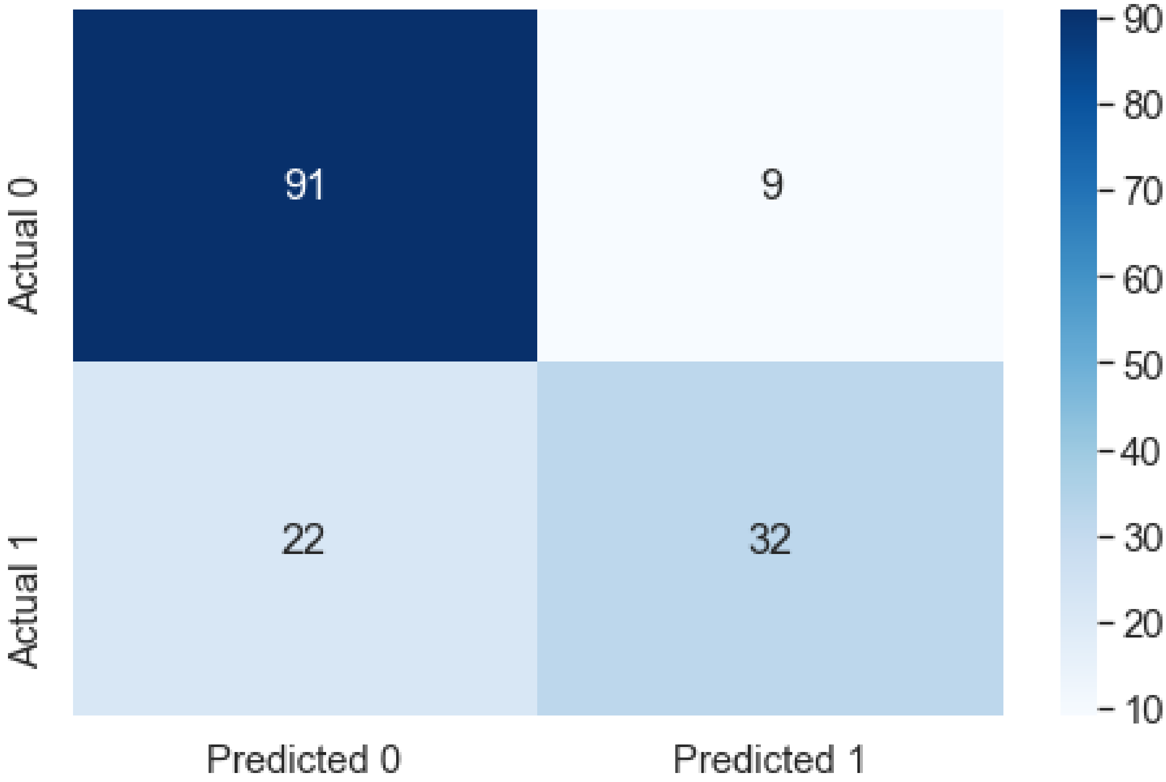
The optimal decision threshold probability is the one that maximizes the K-fold cross validation accuracy.

```
# Optimal decision threshold probability
hyperparam_vals[accuracy_iter.index(max(accuracy_iter))]
```

```
0.46
```

```
# Performance metrics computation for the optimum decision threshold probability
desired_threshold = 0.46

# Classifying observations in the positive class (y = 1) if the predicted probability is g
# than the desired decision threshold probability
y_pred_desired_threshold = y_pred_stratified_prob > desired_threshold
y_pred_desired_threshold = y_pred_desired_threshold.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(y_test_stratified, y_pred_desired_threshold))
print("Recall: ", recall_score(y_test_stratified, y_pred_desired_threshold))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_desired_threshold),
                  columns=['Predicted 0', 'Predicted 1'], index = ['Actual 0', 'Actual 1']
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  79.87012987012987
ROC-AUC:  0.8505555555555556
Precision:  0.7804878048780488
Recall:  0.5925925925925926
```



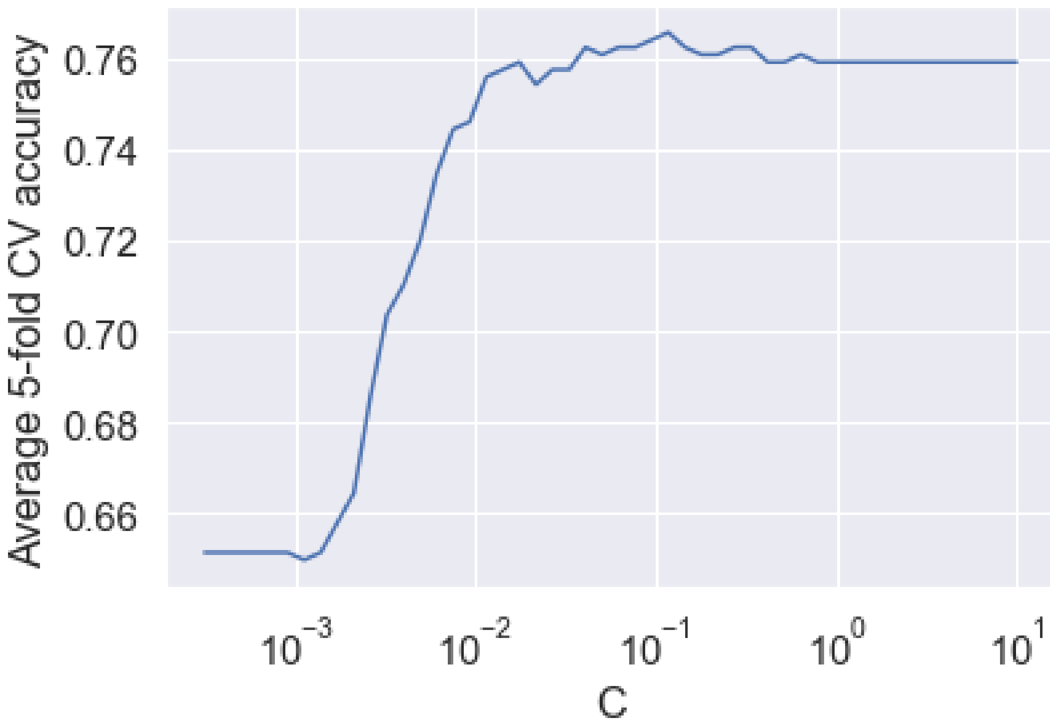
Model performance on test data has improved with the optimal decision threshold probability.

1.5.2 Tuning the regularization parameter

The `LogisticRegression()` method has a default *L2* regularization penalty, which means ridge regression.`C` is $1/\lambda$, where λ is the hyperparameter that is multiplied with the ridge penalty. `C` is 1 by default.

```
accuracy_iter = []
hyperparam_vals = 10**np.linspace(-3.5, 1)

for c_val in hyperparam_vals: # For each possible C value in your grid
    logreg_model = LogisticRegression(C=c_val) # Create a model with the C value

    accuracy_iter.append(cross_val_score(logreg_model, X_train_stratified_scaled, y_train_
                                          scoring='accuracy', cv=5)) # Find the cv results
```

```python
plt.plot(hyperparam_vals, np.mean(np.array(accuracy_iter), axis=1))
plt.xlabel('C')
plt.ylabel('Average 5-fold CV accuracy')
plt.xscale('log')
plt.show()
```



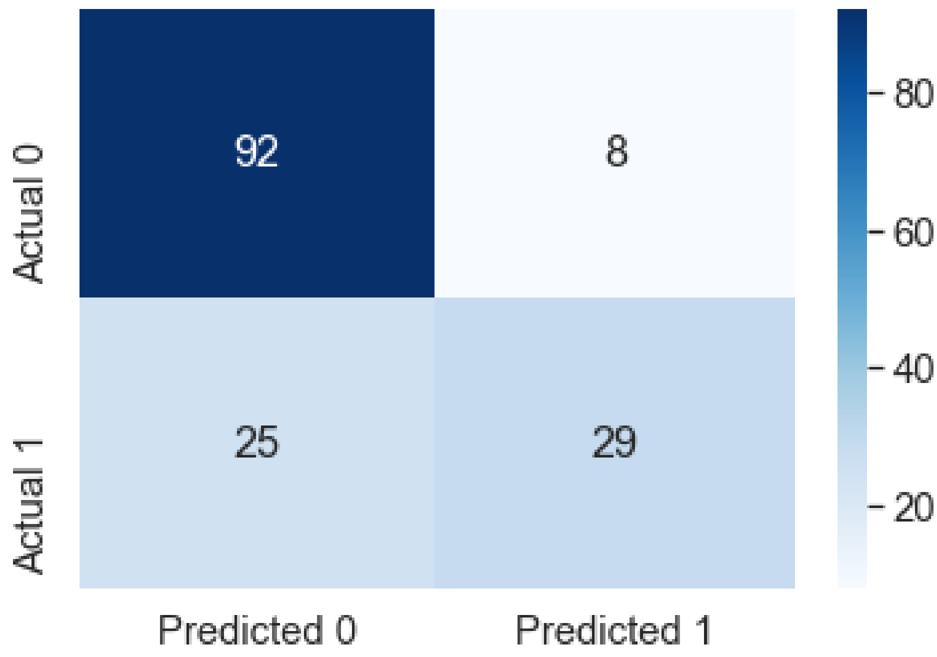

```python
# Optimal value of the regularization parameter 'C'
optimal_C = hyperparam_vals[np.argmax(np.array(accuracy_iter).mean(axis=1))]
optimal_C
```

```
0.11787686347935879
```

```python
# Developing the model with stratified splitting and optimal 'C'

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)
```

```
# Training the model
logreg = LogisticRegression(C = optimal_C)
logreg.fit(X_train_stratified_scaled, y_train_stratified)

#Computing the accuracy
y_pred_stratified = logreg.predict(X_test_stratified_scaled)
print("Accuracy: ",accuracy_score(y_pred_stratified, y_test_stratified)*100)

#Computing the ROC-AUC
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC

#Computing the precision and recall
print("Precision: ", precision_score(y_test_stratified, y_pred_stratified))
print("Recall: ", recall_score(y_test_stratified, y_pred_stratified))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_stratified), columns=['Predic
            index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  78.57142857142857
ROC-AUC:  0.8516666666666666
Precision:  0.7837837837837838
Recall:  0.5370370370370371
```



1.5.3 Tuning the decision threshold probability and the regularization parameter simultaneously

```
threshold_hyperparam_vals = np.arange(0,1.01,0.01)
C_hyperparam_vals = 10**np.linspace(-3.5, 1)
accuracy_iter = pd.DataFrame(columns = {'threshold', 'C', 'accuracy'})
iter_number = 0

for c_val in C_hyperparam_vals:
    predicted_probability = cross_val_predict(LogisticRegression(C = c_val), X_train_strat
                                              y_train_stratified, cv = 5, method = 'pr

    for threshold_prob in threshold_hyperparam_vals:
        predicted_class = predicted_probability[:,1] > threshold_prob
        predicted_class = predicted_class.astype(int)

        #Computing the accuracy
        accuracy = accuracy_score(predicted_class, y_train_stratified)*100
        accuracy_iter.loc[iter_number, 'threshold'] = threshold_prob
        accuracy_iter.loc[iter_number, 'C'] = c_val
        accuracy_iter.loc[iter_number, 'accuracy'] = accuracy
```

```
        iter_number = iter_number + 1
```

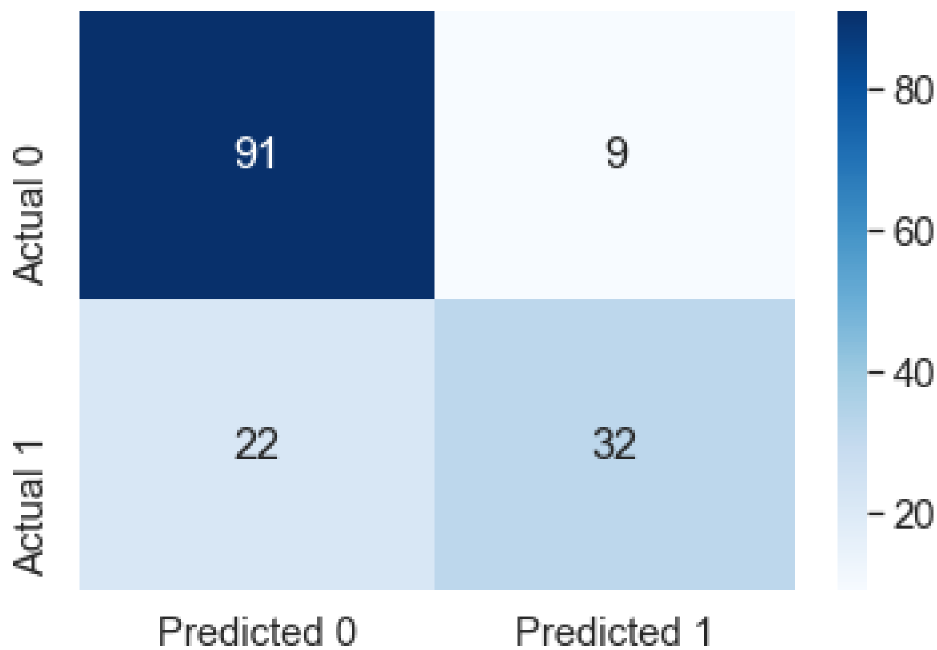
```
# Parameters for highest accuracy
optimal_C = accuracy_iter.sort_values(by = 'accuracy', ascending = False).iloc[0,:]['C']
optimal_threshold = accuracy_iter.sort_values(by = 'accuracy', ascending = False).iloc[0,

#Optimal decision threshold probability
print("Optimal decision threshold = ", optimal_threshold)

#Optimal C
print("Optimal C = ", optimal_C)
```

```
Optimal decision threshold =  0.46
Optimal C =  4.291934260128778
```

```
# Developing the model with stratified splitting, optimal decision threshold probability,

#Scaling data
scaler = StandardScaler().fit(X_train_stratified)
X_train_stratified_scaled = scaler.transform(X_train_stratified)
X_test_stratified_scaled = scaler.transform(X_test_stratified)

# Training the model
logreg = LogisticRegression(C = optimal_C)
logreg.fit(X_train_stratified_scaled, y_train_stratified)

# Performance metrics computation for the optimal threshold probability
y_pred_stratified_prob = logreg.predict_proba(X_test_stratified_scaled)[:,1]

# Classifying observations in the positive class (y = 1) if the predicted probability is g
# than the desired decision threshold probability
y_pred_desired_threshold = y_pred_stratified_prob > optimal_threshold
y_pred_desired_threshold = y_pred_desired_threshold.astype(int)

#Computing the accuracy
print("Accuracy: ",accuracy_score(y_pred_desired_threshold, y_test_stratified)*100)

#Computing the ROC-AUC
fpr, tpr, auc_thresholds = roc_curve(y_test_stratified, y_pred_stratified_prob)
print("ROC-AUC: ",auc(fpr, tpr))# AUC of ROC
```

```
#Computing the precision and recall
print("Precision: ", precision_score(y_test_stratified, y_pred_desired_threshold))
print("Recall: ", recall_score(y_test_stratified, y_pred_desired_threshold))

#Confusion matrix
cm = pd.DataFrame(confusion_matrix(y_test_stratified, y_pred_desired_threshold), columns=[
            index = ['Actual 0', 'Actual 1'])
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g');
```

```
Accuracy:  79.87012987012987
ROC-AUC:  0.8509259259259259
Precision:  0.7804878048780488
Recall:  0.5925925925925926
```



Later in the course, we'll see the `sklearn` function `GridSearchCV`, which is used to optimize several model hyperparameters simultaneously with K-fold cross validation, while avoiding `for` loops.

2 Regression splines

Read sections 7.1-7.4 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
from patsy import dmatrix
from sklearn.metrics import mean_squared_error
from pyearth import Earth
from sklearn.linear_model import LinearRegression
```

```
#Using the same datasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('./Datasets/Car_features_train.csv')
trainp = pd.read_csv('./Datasets/Car_prices_train.csv')
testf = pd.read_csv('./Datasets/Car_features_test.csv')
testp = pd.read_csv('./Datasets/Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

2.1 Polynomial regression vs Regression splines

2.1.1 Model of degree 1

```
X = pd.DataFrame(train['mileage'])
X_test = pd.DataFrame(test['mileage'])
y = train['price']
lr_model = LinearRegression()
lr_model.fit(X, y);
```

```
#Regression spline of degree 1

#Creating basis functions for splines of degree 1
transformed_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_inte
                        data = trainf,return_type = 'dataframe')
```

```
transformed_x.head()
```

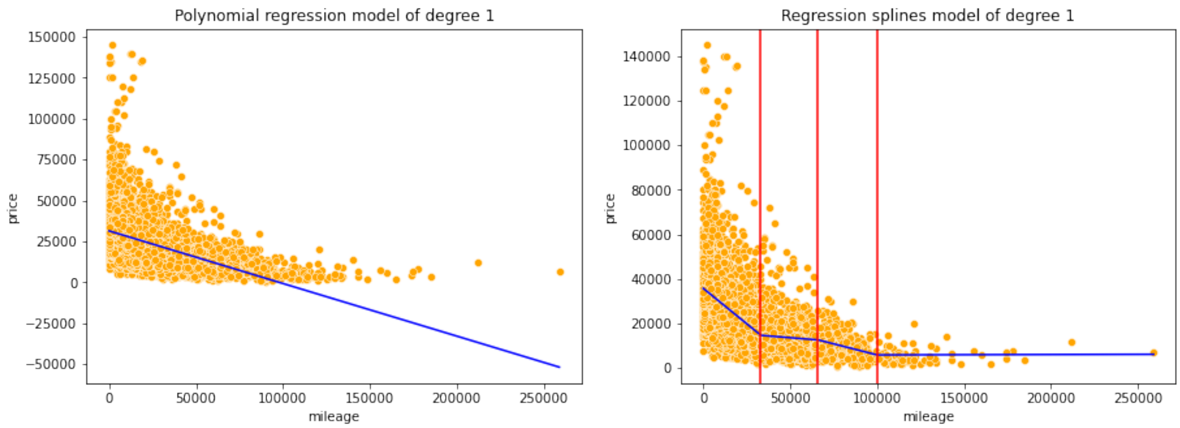
	Intercept	bs(mileage, knots=(33000, 66000, 100000), degree=1, include_intercept=False)[0]	bs(mileage,
0	1.0	0.000303	0.000000
1	1.0	0.327646	0.000000
2	1.0	0.000152	0.000000
3	1.0	0.572563	0.000000
4	1.0	0.092333	0.907667

Note that the truncated power basis in the class presentation is conceptually simple to understand, it may run into numerical issues as powers of large numbers can lead to severe rounding errors. The `bs()` function generates the B-sline basis, which allows for efficient computation, especially in case of a large number of knots. All the basis function values are normalized to be in [0, 1] in the B-spline basis. Although we'll use the B-spline basis functions to fit splines, details regarding the B-spline basis functions are not included in the syllabus.

We actually don't need to separately generate basis functions, and then fit the model. We can do it in the same line of code using the `statsmodels` `OLS` method.

```
# Regression spline model with linear splines
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 1, in
                           data = train).fit()
```

```
#Visualizing polynomial model and the regression spline model of degree 1

knots = [33000,66000,100000] #Knots for the spline
d=1 #Degree of predictor in the model
#Writing a function to visualize polynomial model and the regression spline model of degre
def viz_models():
    fig, axes = plt.subplots(1,2,figsize = (15,5))
    plt.subplots_adjust(wspace=0.2)

    #Visualizing the linear regression model
    pred_price = lr_model.predict(X)
    sns.scatterplot(ax = axes[0],x = 'mileage', y = 'price', data = train, color = 'orange
    sns.lineplot(ax = axes[0],x = train.mileage, y = pred_price, color = 'blue')
    axes[0].set_title('Polynomial regression model of degree '+str(d))

    #Visualizing the regression splines model of degree 'd'
    axes[1].set_title('Regression splines model of degree '+ str(d))
    sns.scatterplot(ax=axes[1],x = 'mileage', y = 'price', data = train, color = 'orange')
    sns.lineplot(ax=axes[1],x = train.mileage, y = reg_spline_model.predict(), color = 'bl
    for i in range(3):
        plt.axvline(knots[i], 0,100,color='red')
viz_models()
```



We observe the regression splines model better fits the data as compared to the polynomial regression model. This is because regression splines of degree 1 fit piecewise polynomials, or linear models on sub-sections of the predictor, which helps better capture the trend. However, this added flexibility may also lead to overfitting. Hence, one must be careful to check for overfitting when using splines. Overfitting may be checked by k-fold cross validation or comparing test and train errors.

The red lines in the plot on the right denote the position of knots. Knots separate distinct splines.

Although, we can separately generate the basis functions for test data, it may lead to incaccurate results if the distribution of the predictor values in test data is different from that in the train data. This is because the B-spline basis functions of train data are generated after normalizing the predictor values. If the basis functions of test data are generated independently, their values may be inaccurate, as they will depend on the domain space spanned by the test data.

```
# Basis functions for test data - avoid generating basis functions separately for test dat
# as the test data normailization may be different from the train data normalization
test_x = dmatrix("bs(mileage , knots=(33000,66000,100000), degree = 1, include_intercept =
```

```
#Function to compute RMSE (root mean squared error on train and test datasets)
def rmse():
    #Error on train data for the linear regression model
    print("RMSE on train data:")
    print("Linear regression:", np.sqrt(mean_squared_error(lr_model.predict(X),train.price

    #Error on train data for the regression spline model
    print("Regression splines:", np.sqrt(mean_squared_error(reg_spline_model.predict(X),tr

    #Error on test data for the linear regression model
    print("\nRMSE on test data:")
    print("Linear regression:",np.sqrt(mean_squared_error(lr_model.predict(X_test),test.pr

    #Error on test data for the regression spline model
    print("Regression splines:",np.sqrt(mean_squared_error(reg_spline_model.predict(X_test
rmse()
```

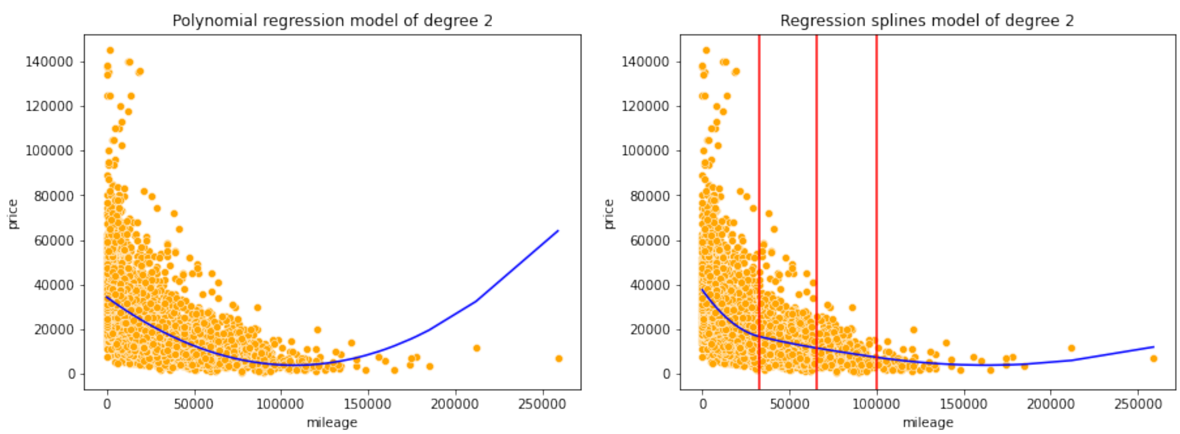
```
RMSE on train data:
Linear regression: 14403.250083261853
Regression splines: 13859.640716531134

RMSE on test data:
Linear regression: 14370.94086395544
Regression splines: 13770.118474361932
```

2.1.2 Model of degree 2

A higher degree model will lead to additional flexibility for both polynomial and regression splines models.

```
#Including mileage squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)', data = train)
lr_model = ols_object.fit()

#Regression spline of degree 2
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 2, in
                           data = train).fit()
```

```
d=2
viz_models()
```



Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

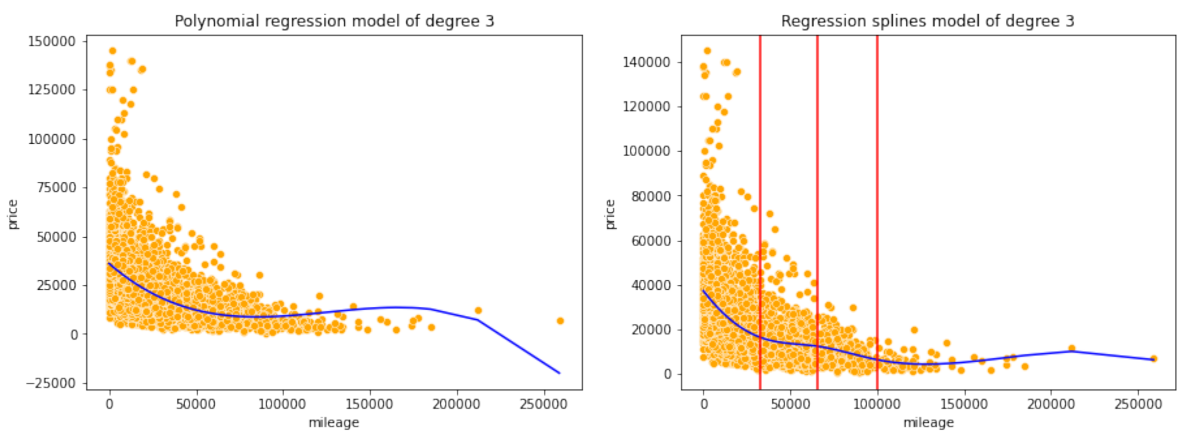
```
rmse()
```

```
RMSE on train data:
Linear regression: 14009.819556665143
Regression splines: 13818.572654146721

RMSE on test data:
Linear regression: 13944.20691909441
Regression splines: 13660.777953039395
```

2.1.3 Model of degree 3

```
#Including mileage cube squared as a predictor and developing the model
ols_object = smf.ols(formula = 'price~mileage+I(mileage**2)+I(mileage**3)', data = train)
lr_model = ols_object.fit()

#Regression spline of degree 3
reg_spline_model = smf.ols('price~bs(mileage, knots = (33000,66000,100000), degree = 3, in
                           data = train).fit()
```

```
d=3
viz_models()
```



Unlike polynomial regression, splines functions avoid imposing a global structure on the non-linear function of X. This provides a better local fit to the data.

```
rmse()
```

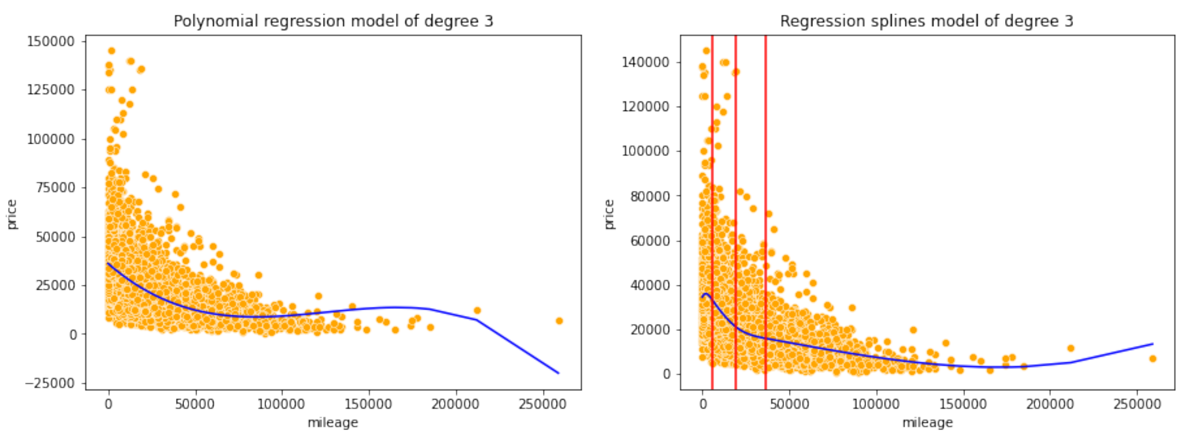
```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13822.70511947823

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13683.776494331632
```

2.2 Regression splines with knots at uniform quantiles of data

If degrees of freedom are provided instead of knots, the knots are by default chosen at uniform quantiles of data. For example if there are 7 degrees of freedom (including the intercept), then there will be 7-4 = 3 knots. These knots will be chosen at the 255h, 50th and 75th quantiles of the data.

```
#Regression spline of degree 3

#Regression spline of degree 3 with knots at uniform quantiles of data
reg_spline_model = smf.ols('price~bs(mileage, df = 6, degree = 3, include_intercept = Fals
                              data = train).fit()
```

```
d=3
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```



Splines can be unstable at the outer range of predictors. Note that splines are themselves piecewise polynomials with no constraints at the 2 extreme ends of the predictor space. Thus, they may become unstable at those 2 ends. In the right scatter plot, we can see that price has a decreasing trend with mileage. However on the extreme left of the plot, we see the trend reversing with regard to the model, which suggests potential overfitting. Also, from the domain knowledge about cars we know that there is no reason why price will reduce if the car is relatively new. Thus, there may be overfitting with cubic splines at / near the extreme points of the domain space. In the figure (on the right), the left-most spline may be overfitting.

This motivates us to introduce natural cubic splines (below), which help with the stability at extreme points by enforcing the spline to be linear at those points. We may also think about it as another kind of a "knot" being put at the two ends to make the spline stable at these points.

```
rmse()
```

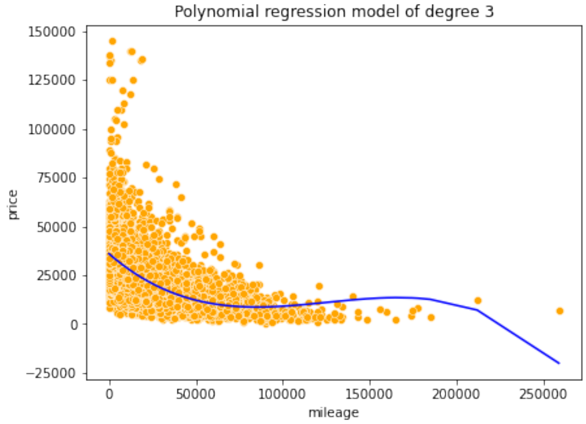
```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13781.79102252679

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13605.726076704668
```

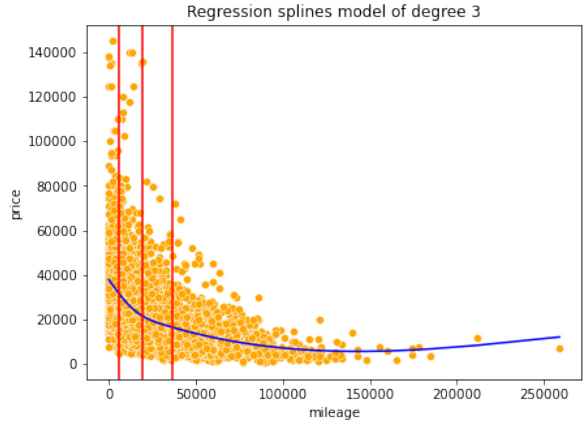
2.3 Natural cubic splines

Page 298: "A natural spline is a regression spline with additional boundary constraints: the function is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot). This additional constraint means that natural splines generally produce more stable estimates at the boundaries."

```
#Natural cubic spline

#Creating basis functions for the natural cubic spline
reg_spline_model = smf.ols('price~cr(mileage, df = 4)',
                           data = train).fit()
```

```
d=3;
unif_knots = pd.qcut(train.mileage,4,retbins=True)[1][1:4]
knots=unif_knots
viz_models()
```





Note that the natural cubic spline is more stable than a cubic splines with knots at uniformly distributed quantiles.

```
rmse()
```

```
RMSE on train data:
Linear regression: 13891.962447594644
Regression splines: 13826.125469174143

RMSE on test data:
Linear regression: 13789.708418357186
Regression splines: 13660.35327661836
```

2.4 Generalized additive model (GAM)

GAM allow for flexible nonlinearities in several variables, but retain the additive structure of linear models. In a GAM, non-linear basis functions of predictors can be used as predictors of a linear regression model. For example,

$$y = f_1(X_1) + f_2(X_2) + \epsilon$$

is a GAM, where $f_1(.)$ may be a cubic spline based on the predictor X_1, and $f_2(.)$ may be a step function based on the predictor X_2.

```
#GAM
#GAM includes cubic splines for mileage. Other predictors are year, engineSize, mpg, milea
model_gam = smf.ols('price~bs(mileage,df=6,degree = 3)+year*engineSize*mpg*mileage', data
```

```
preds = model_gam.predict(test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
8393.773177637542
```

```
#GAM
#GAM includes cubic splines for mileage, year, engineSize, mpg, and interactions of all pr
model_gam = smf.ols('price~bs(mileage,df=6,degree = 3)+bs(mpg,df=6,degree = 3)+\
bs(engineSize,df=6,degree = 3)+year*engineSize*mpg*mileage', data = train).fit()

preds = model_gam.predict(test)
np.sqrt(mean_squared_error(preds,test.price))
```

```
7981.100853841914
```

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2+I(mileage**2)+I(mil
model = ols_object.fit()
model.summary()
```

Table 2.3: OLS Regression Results

Dep. Variable:	price	R-squared:	0.704
Model:	OLS	Adj. R-squared:	0.703
Method:	Least Squares	F-statistic:	1308.
Date:	Sun, 09 Apr 2023	Prob (F-statistic):	0.00
Time:	20:48:35	Log-Likelihood:	-52157.
No. Observations:	4960	AIC:	1.043e+05
Df Residuals:	4950	BIC:	1.044e+05
Df Model:	9		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-0.0009	0.000	-2.752	0.006	-0.002	-0.000
year	-1.1470	0.664	-1.728	0.084	-2.448	0.154
engineSize	0.0052	0.000	17.419	0.000	0.005	0.006
mileage	-31.4751	2.621	-12.010	0.000	-36.613	-26.337
mpg	-0.0201	0.002	-13.019	0.000	-0.023	-0.017
year:engineSize	9.5957	0.254	37.790	0.000	9.098	10.094

year:mileage	0.0154	0.001	11.816	0.000	0.013	0.018
year:mpg	0.0572	0.013	4.348	0.000	0.031	0.083
engineSize:mileage	-0.1453	0.008	-18.070	0.000	-0.161	-0.130
engineSize:mpg	-98.9062	11.832	-8.359	0.000	-122.102	-75.710
mileage:mpg	0.0011	0.000	2.432	0.015	0.000	0.002
I(mileage ** 2)	7.713e-06	3.75e-07	20.586	0.000	6.98e-06	8.45e-06
I(mileage ** 3)	-1.867e-11	1.43e-12	-13.077	0.000	-2.15e-11	-1.59e-11

Omnibus:	1830.457	Durbin-Watson:	0.634
Prob(Omnibus):	0.000	Jarque-Bera (JB):	34927.811
Skew:	1.276	Prob(JB):	0.00
Kurtosis:	15.747	Cond. No.	2.50e+18

```
np.sqrt(mean_squared_error(model.predict(test),test.price))
```

```
9026.775740000594
```

Note the RMSE with GAM that includes regression splines for mileage is lesser than that of the linear regression model, indicating a better fit.

2.5 MARS (Multivariate Adaptive Regression Splines)

```
from pyearth import Earth
X=train['mileage']
y=train['price']
```

2.5.1 MARS of degree 1

```
model = Earth(max_terms=500, max_degree=1) # note, terms in brackets are the hyperparamete
model.fit(X,y)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
Earth(max_degree=1, max_terms=500)
```

```
print(model.summary())
```

```
Earth Model
-------------------------------------
Basis Function  Pruned  Coefficient
-------------------------------------
(Intercept)     No      -553155
h(x0-22141)     Yes     None
h(22141-x0)     Yes     None
h(x0-3354)      No      -6.23571
h(3354-x0)      Yes     None
h(x0-15413)     No      -36.9613
h(15413-x0)     No      38.167
h(x0-106800)    Yes     None
h(106800-x0)    No      0.221844
h(x0-500)       No      170.039
h(500-x0)       Yes     None
h(x0-741)       Yes     None
h(741-x0)       No      -54.5265
h(x0-375)       No      -126.804
h(375-x0)       Yes     None
h(x0-2456)      Yes     None
h(2456-x0)      No      7.04609
-------------------------------------
MSE: 188429705.7549, GCV: 190035470.5664, RSQ: 0.2998, GRSQ: 0.2942
```

Model equation:

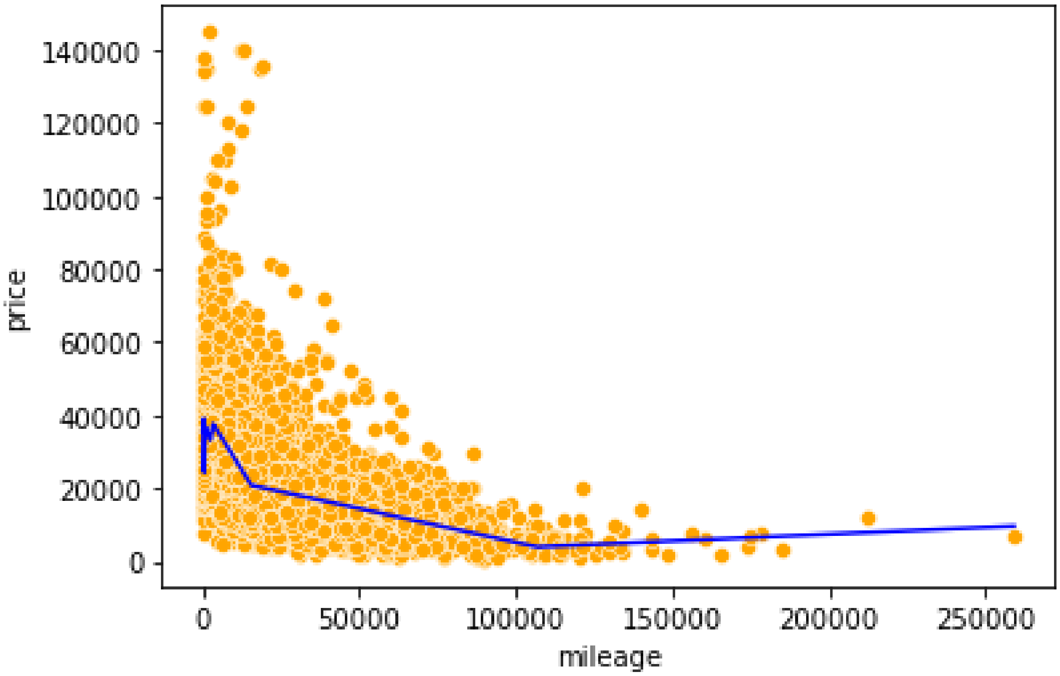
$$-553155 - 6.23(h(x0 - 3354)) - 36.96(h(x0 - 15413) + ... - 7.04(h(2456 - x0)$$

```
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13650.2113154515
```

```
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



2.5.2 MARS of degree 2

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
Earth Model
-----------------------------------------------
Basis Function          Pruned  Coefficient
-----------------------------------------------
(Intercept)             No      19369.7
h(x0-22141)             Yes     None
h(22141-x0)             Yes     None
```

```
h(x0-7531)*h(22141-x0)   No      3.74934e-05
h(7531-x0)*h(22141-x0)   No      -6.74252e-05
x0*h(x0-22141)           No      -8.0703e-06
h(x0-15012)              Yes     None
h(15012-x0)              No      1.79813
h(x0-26311)*h(x0-22141)  No      8.85097e-06
h(26311-x0)*h(x0-22141)  Yes     None
-----------------------------------------------
MSE: 189264421.5682, GCV: 190298913.1652, RSQ: 0.2967, GRSQ: 0.2932
```

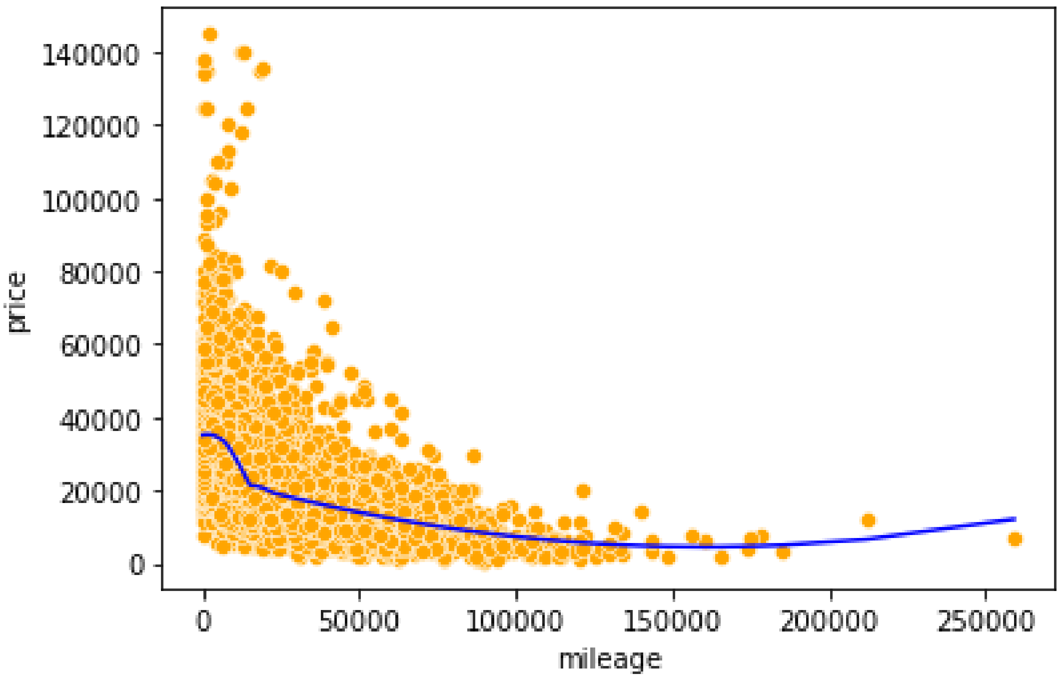
```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  pruning_passer.run()
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```python
pred = model.predict(test.mileage)
np.sqrt(mean_squared_error(pred,test.price))
```

```
13590.995419204985
```

```python
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = train.mileage, y = model.predict(train.mileage), color = 'blue')
```

```
<AxesSubplot:xlabel='mileage', ylabel='price'>
```



MARS provides a better fit than the splines that we used above. This is because MARS tunes the positions of the knots and considers interactions (also with tuned knots) to improve the model fit. Tuning of knots may improve the fit of splines as well.

2.5.3 MARS including categorical variables

```
#A categorical variable can be turned to dummy variables to use the Earth package for fitt
train_cat = pd.get_dummies(train)
test_cat = pd.get_dummies(test)
```

```
train_cat.head()
```

	carID	year	mileage	tax	mpg	engineSize	price	brand_audi	brand_bmw	brand_ford	…
0	18473	2020	11	145	53.3282	3.0	37980	0	1	0	…
1	15064	2019	10813	145	53.0430	3.0	33980	0	1	0	…
2	18268	2020	6	145	53.4379	3.0	36850	0	1	0	…
3	18480	2017	18895	145	51.5140	3.0	25998	0	1	0	…
4	18492	2015	62953	160	51.4903	3.0	18990	0	1	0	…

```python
X = train_cat[['mileage','mpg','engineSize','year','fuelType_Diesel','fuelType_Electric',
               'fuelType_Hybrid','fuelType_Petrol']]
Xtest = test_cat[['mileage','mpg','engineSize','year','fuelType_Diesel','fuelType_Electric
                  'fuelType_Hybrid','fuelType_Petrol']]
```

```python
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
```

```
Earth Model
----------------------------------------------------------
Basis Function                       Pruned  Coefficient
----------------------------------------------------------
(Intercept)                          No      2.17604e+06
h(engineSize-5.5)                    No      9.80752e+06
h(5.5-engineSize)                    No      1.92817e+06
h(mileage-21050)                     No      18.687
h(21050-mileage)                     No      -177.871
h(mileage-21050)*h(5.5-engineSize)   Yes     None
h(21050-mileage)*h(5.5-engineSize)   No      -0.224909
year                                 No      4126.41
h(mpg-53.3495)                       No      344595
h(53.3495-mpg)                       Yes     None
fuelType_Hybrid*h(5.5-engineSize)    No      6124.34
h(mileage-21050)*year                No      -0.00930239
h(21050-mileage)*year                No      0.0886455
h(engineSize-5.5)*year               No      -4864.84
h(5.5-engineSize)*year               No      -952.92
h(mileage-1422)*h(53.3495-mpg)       No      -16.62
h(1422-mileage)*h(53.3495-mpg)       No      16.4306
fuelType_Hybrid                      No      -89090.6
h(mpg-21.1063)*h(53.3495-mpg)        Yes     None
h(21.1063-mpg)*h(53.3495-mpg)        No      -8815.99
h(mpg-23.4808)*h(5.5-engineSize)     No      -3649.97
h(23.4808-mpg)*h(5.5-engineSize)     Yes     None
h(mpg-20.5188)*year                  No      31.7341
h(20.5188-mpg)*year                  Yes     None
```

```
h(mpg-22.2566)*h(53.3495-mpg)       No      -52.2531
h(22.2566-mpg)*h(53.3495-mpg)       No      7916.19
h(mpg-22.6767)                      No      7.56432e+06
h(22.6767-mpg)                      Yes     None
h(mpg-23.9595)*h(mpg-22.6767)       Yes     None
h(23.9595-mpg)*h(mpg-22.6767)       No      -63225.4
h(mpg-21.4904)*h(22.6767-mpg)       No      -149055
h(21.4904-mpg)*h(22.6767-mpg)       Yes     None
h(mpg-21.1063)                      No      -887098
h(21.1063-mpg)                      Yes     None
h(mpg-29.5303)*h(mpg-22.6767)       No      -3028.87
h(29.5303-mpg)*h(mpg-22.6767)       Yes     None
h(mpg-28.0681)*h(5.5-engineSize)    No      3572.89
h(28.0681-mpg)*h(5.5-engineSize)    Yes     None
engineSize*h(5.5-engineSize)        No      -2952.65
h(mpg-25.3175)*h(mpg-21.1063)       No      -332551
h(25.3175-mpg)*h(mpg-21.1063)       No      324298
fuelType_Petrol*year                No      -1.37031
h(mpg-68.9279)*fuelType_Hybrid      No      -4087.9
h(68.9279-mpg)*fuelType_Hybrid      Yes     None
h(mpg-31.5043)*h(5.5-engineSize)    Yes     None
h(31.5043-mpg)*h(5.5-engineSize)    No      3691.82
h(mpg-32.7011)*h(5.5-engineSize)    Yes     None
h(32.7011-mpg)*h(5.5-engineSize)    No      -2262.78
h(mpg-44.9122)*h(mpg-22.6767)       No      335577
h(44.9122-mpg)*h(mpg-22.6767)       No      -335623
h(engineSize-5.5)*h(mpg-21.1063)    No      27815
h(5.5-engineSize)*h(mpg-21.1063)    Yes     None
h(mpg-78.1907)*fuelType_Hybrid      Yes     None
h(78.1907-mpg)*fuelType_Hybrid      No      2221.49
h(mpg-63.1632)*h(mpg-22.6767)       Yes     None
h(63.1632-mpg)*h(mpg-22.6767)       No      21.0093
fuelType_Hybrid*h(mpg-53.3495)      No      4121.91
h(mileage-22058)*h(53.3495-mpg)     No      16.6177
h(22058-mileage)*h(53.3495-mpg)     No      -16.6044
h(mpg-21.8985)                      Yes     None
h(21.8985-mpg)                      No      371659
---------------------------------------------------------
MSE: 45859836.5623, GCV: 47884649.3622, RSQ: 0.8296, GRSQ: 0.8221

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(pred,test.price))
```

```
7499.709075454322
```

Let us compare the RMSE of a MARS model with *mileage, mpg, engineSize* and *year* with a linear regression model having the same predictors.

```
X = train[['mileage','mpg','engineSize','year']]
```

```
model = Earth(max_terms=500, max_degree=2) # note, terms in brackets are the hyperparamete
model.fit(X,y)
print(model.summary())
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:813: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
  pruning_passer.run()
```

```
Earth Model
----------------------------------------------------------
Basis Function                       Pruned  Coefficient
----------------------------------------------------------
(Intercept)                          No      -8.13682e+06
h(engineSize-5.5)                    No      9.53908e+06
h(5.5-engineSize)                    Yes     None
h(mileage-21050)                     No      23.4448
h(21050-mileage)                     No      -215.861
h(mileage-21050)*h(5.5-engineSize)   Yes     None
h(21050-mileage)*h(5.5-engineSize)   No      -0.278562
year                                 No      4125.85
h(mpg-53.3495)                       Yes     None
h(53.3495-mpg)                       Yes     None
h(mileage-21050)*year                No      -0.0116601
h(21050-mileage)*year                No      0.107624
h(mpg-53.2957)*h(5.5-engineSize)     No      -59801.3
h(53.2957-mpg)*h(5.5-engineSize)     No      59950.5
h(engineSize-5.5)*year               No      -4713.74
h(5.5-engineSize)*year               No      -755.742
h(mileage-1766)*h(53.3495-mpg)       No      -0.00337072
h(1766-mileage)*h(53.3495-mpg)       No      -0.144905
```

```
h(mpg-19.1277)*h(53.3495-mpg)       No      161.153
h(19.1277-mpg)*h(53.3495-mpg)       Yes     None
h(mpg-23.4808)*h(5.5-engineSize)    Yes     None
h(23.4808-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-21.4971)*h(5.5-engineSize)    Yes     None
h(21.4971-mpg)*h(5.5-engineSize)    Yes     None
h(mpg-40.224)*h(5.5-engineSize)     Yes     None
h(40.224-mpg)*h(5.5-engineSize)     No      298.139
engineSize*h(5.5-engineSize)        No      -2553.17
h(mpg-22.2566)                      Yes     None
h(22.2566-mpg)                      No      29257.3
h(mpg-20.7712)*h(22.2566-mpg)       No      143796
h(20.7712-mpg)*h(22.2566-mpg)       No      -1249.17
h(mpg-21.4971)*h(22.2566-mpg)       No      -315486
h(21.4971-mpg)*h(22.2566-mpg)       Yes     None
h(mpg-27.0995)*h(mpg-22.2566)       No      3855.71
h(27.0995-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-29.3902)*year                 No      6.05449
h(29.3902-mpg)*year                 No      -20.176
h(mpg-28.0681)*h(5.5-engineSize)    No      59901.6
h(28.0681-mpg)*h(5.5-engineSize)    No      -55502.2
h(mpg-23.2962)*h(mpg-22.2566)       No      -56126
h(23.2962-mpg)*h(mpg-22.2566)       No      73153.9
h(mpg-69.0719)*h(mpg-53.3495)       Yes     None
h(69.0719-mpg)*h(mpg-53.3495)       No      -124.847
h(engineSize-5.5)*h(22.2566-mpg)    No      -20955.8
h(5.5-engineSize)*h(22.2566-mpg)    No      -8336.23
h(mpg-23.9595)*h(mpg-22.2566)       No      -62983
h(23.9595-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-23.6406)*h(mpg-22.2566)       No      115253
h(23.6406-mpg)*h(mpg-22.2566)       Yes     None
h(mpg-56.1908)                      Yes     None
h(56.1908-mpg)                      No      -2239.85
h(mpg-29.7993)*h(53.3495-mpg)       No      -139.61
h(29.7993-mpg)*h(53.3495-mpg)       No      788.756
----------------------------------------------------------
MSE: 49704412.0771, GCV: 51526765.3943, RSQ: 0.8153, GRSQ: 0.8086

C:\Users\akl0407\Anaconda3\lib\site-packages\pyearth\earth.py:1066: FutureWarning: `rcond` pa
To use the future default and silence this warning we advise to pass `rcond=None`, to keep us
  coef, resid = np.linalg.lstsq(B, weighted_y[:, i])[0:2]
```

```
Xtest = test[['mileage','mpg','engineSize','year']]
pred = model.predict(Xtest)
np.sqrt(mean_squared_error(pred,test.price))
```

```
7614.158359050244
```

```
ols_object = smf.ols(formula = 'price~(year+engineSize+mileage+mpg)**2', data = train)
model = ols_object.fit()
pred = model.predict(test)
np.sqrt(mean_squared_error(pred,test.price))
```

```
8729.912066822455
```

The RMSE for the MARS model is lesser than that of the linear regression model, as expected.

3 Regression trees

Read section 8.1.1 of the book before using these notes.

Note that in this course, lecture notes are not sufficient, you must read the book for better understanding. Lecture notes are just implementing the concepts of the book on a dataset, but not explaining the concepts elaborately.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score,train_test_split
from sklearn.metrics import mean_squared_error,r2_score
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV, ParameterGrid

from sklearn.tree import export_graphviz
from six import StringIO
from IPython.display import Image
import pydotplus
import time as time
```

```
#Using the same datsasets as used for linear regression in STAT303-2,
#so that we can compare the non-linear models with linear regression
trainf = pd.read_csv('Car_features_train.csv')
trainp = pd.read_csv('Car_prices_train.csv')
testf = pd.read_csv('Car_features_test.csv')
testp = pd.read_csv('Car_prices_test.csv')
train = pd.merge(trainf,trainp)
test = pd.merge(testf,testp)
train.head()
```

	carID	brand	model	year	transmission	mileage	fuelType	tax	mpg	engineSize	price
0	18473	bmw	6 Series	2020	Semi-Auto	11	Diesel	145	53.3282	3.0	37980
1	15064	bmw	6 Series	2019	Semi-Auto	10813	Diesel	145	53.0430	3.0	33980
2	18268	bmw	6 Series	2020	Semi-Auto	6	Diesel	145	53.4379	3.0	36850
3	18480	bmw	6 Series	2017	Semi-Auto	18895	Diesel	145	51.5140	3.0	25998
4	18492	bmw	6 Series	2015	Automatic	62953	Diesel	160	51.4903	3.0	18990

3.1 Building a regression tree

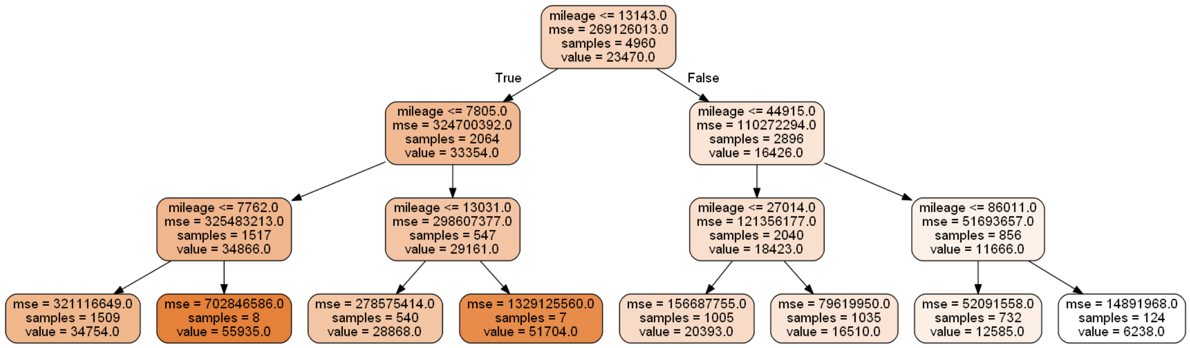
Develop a regression tree to predict car price based on mileage

```
X = train['mileage']
y = train['price']
```

```
#Defining the object to build a regression tree
model = DecisionTreeRegressor(random_state=1, max_depth=3)

#Fitting the regression tree to the data
model.fit(X.values.reshape(-1,1), y)
```

```
DecisionTreeRegressor(max_depth=3, random_state=1)
```

```
#Visualizing the regression tree
dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True, rounded=True,
                feature_names =['mileage'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



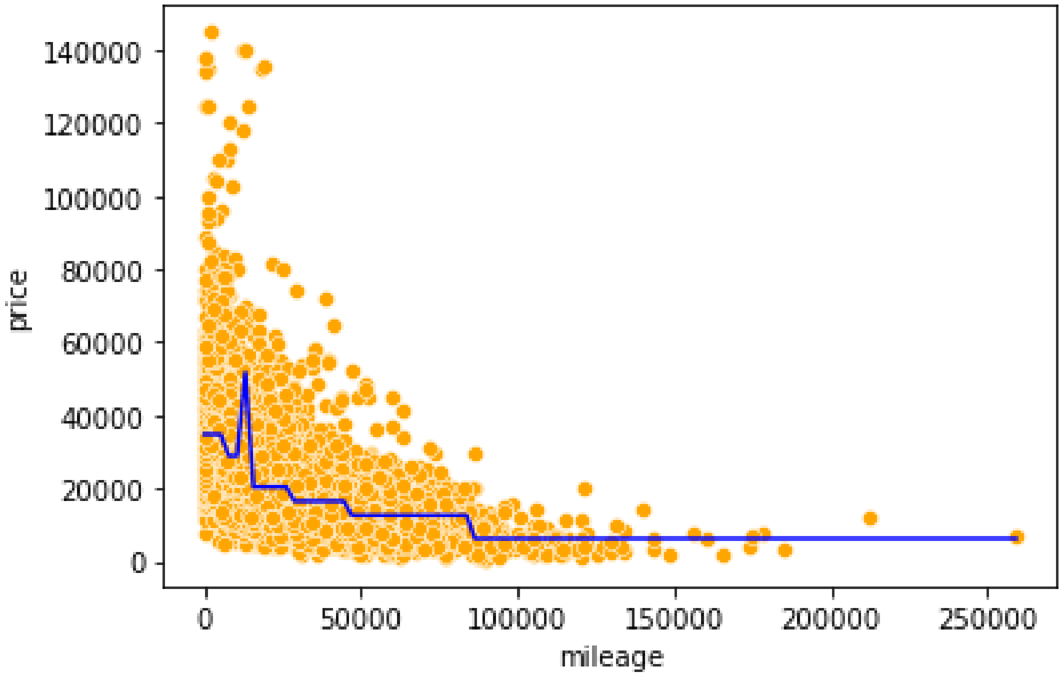

```python
#prediction on test data
pred=model.predict(test[['mileage']])
```

```python
#RMSE on test data
np.sqrt(mean_squared_error(test.price, pred))
```

13764.798425410803

```python
#Visualizing the model fit
Xtest = np.linspace(min(X), max(X), 100)
pred_test = model.predict(Xtest.reshape(-1,1))
sns.scatterplot(x = 'mileage', y = 'price', data = train, color = 'orange')
sns.lineplot(x = Xtest, y = pred_test, color = 'blue')
```

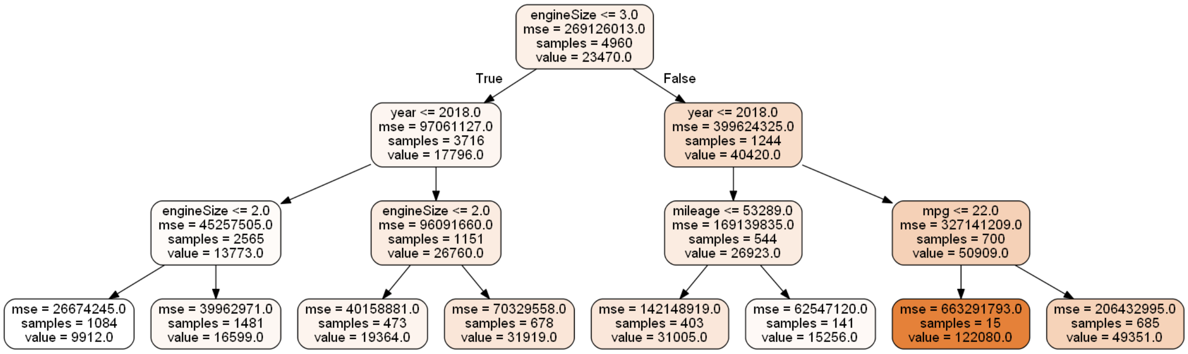
<AxesSubplot:xlabel='mileage', ylabel='price'>



All cars falling within the same terminal node have the same predicted price, which is seen as flat line segments in the above model curve

Develop a regression tree to predict car price based on mileage, mpg, engineSize and year

```
X = train[['mileage','mpg','year','engineSize']]
model = DecisionTreeRegressor(random_state=1, max_depth=3)
model.fit(X, y)
dot_data = StringIO()
export_graphviz(model, out_file=dot_data,
                filled=True, rounded=True,
                feature_names =['mileage','mpg','year','engineSize'],precision=0)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('car_price_tree.png')
Image(graph.create_png())
```



3.2 Optimizing parameters to improve the regression tree

Let us find the optimal depth of the tree and the number of terminal nods (leaves) by cross validation.

```
#Finding cross validation error for trees ranging from a depth of 1 to 19.
parameters = {'max_depth':range(3,20),'max_leaf_nodes':range(100,300)}
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
model = GridSearchCV(DecisionTreeRegressor(random_state=1), parameters, n_jobs=-1,verbose=
model.fit(X, y)
print (model.best_score_, model.best_params_)
```

```
Fitting 5 folds for each of 3400 candidates, totalling 17000 fits
0.8465176078797111 {'max_depth': 10, 'max_leaf_nodes': 262}
```

```
#Detailed results of k-fold cross validation
pd.DataFrame(model.cv_results_).head()
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	param_max
0	0.006249	0.007653	0.009373	0.007653	3	100
1	0.012497	0.006248	0.003124	0.006248	3	101
2	0.015622	0.000002	0.000000	0.000000	3	102
3	0.012497	0.006248	0.012496	0.006248	3	103
4	0.015622	0.000001	0.000000	0.000000	3	104

```
#Developing the tree based on optimal parameters found by cross-validation
model = DecisionTreeRegressor(random_state=1, max_depth=10,max_leaf_nodes=262)
```

```
model.fit(X, y)
```

```
DecisionTreeRegressor(max_depth=10, max_leaf_nodes=262, random_state=1)
```

```
#RMSE on test data
Xtest = test[['mileage','mpg','year','engineSize']]
np.sqrt(mean_squared_error(test.price, model.predict(Xtest)))
```

```
6921.0404660552895
```

The RMSE for the decision tree is lower than that of linear regression models and spline regression models (including MARS), with these four predictors. This may be probably due to car price having a highly non-linear association with the predictors.

Predictor importance: The importance of a predictor is computed as the (normalized) total reduction of the criterion (SSE in case of regression trees) brought by that predictor.

Warning: impurity-based feature importances can be misleading for high cardinality features (many unique values) *Source: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegres*

```
model.feature_importances_
```

```
array([0.04490344, 0.15882336, 0.29739951, 0.49887369])
```

Engine size is the most important predictor, followed by *year*, which is followed by *mpg*, and *mileage* is the least important predictor.

3.3 Cost complexity pruning

While optimizing parameters above, we optimized them within a range that we thought was reasonable. While doing so, we restricted ouverselves to considering only a subset of the unpruned tree. Thus, we could have missed out on finding the optimal tree (or the best model).

With cost complexity pruning, we first develop an unpruned tree without any restrictions. Then, using cross validation, we find the optimal value of the tuning parameter α. All the non-terminal nodes for which α_{eff} is smaller that the optimal α will be pruned. You will need to check out the link below to understand this better.

Check out a detailed explanation of how cost complexity pruning is implemented in sklearn at: https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning

Here are some informative visualizations that will help you understand what is happening in cost complexity pruning: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.h glr-auto-examples-tree-plot-cost-complexity-pruning-py

```
model = DecisionTreeRegressor(random_state = 1)#model without any restrictions
path= model.cost_complexity_pruning_path(X,y)# Compute the pruning path during Minimal Cos
```

```
alphas=path['ccp_alphas']
```

```
len(alphas)
```

```
4126
```

```
start_time = time.time()
cv = KFold(n_splits = 5,shuffle=True,random_state=1)
tree = GridSearchCV(DecisionTreeRegressor(random_state=1), param_grid = {'ccp_alpha':alpha
                     scoring = 'neg_mean_squared_error',n_jobs=-1,verbose=1,cv=cv)
tree.fit(X, y)
print (tree.best_score_, tree.best_params_)
total_time = time.time()-start_time
```

```
Fitting 5 folds for each of 4126 candidates, totalling 20630 fits
-44150619.209031895 {'ccp_alpha': 143722.94076639024}
```

```
total_time/60
```

```
2.332933847109477
```

The code took 2 minutes to run on a dataset of about 5000 observations and 4 predictors.

```
tree = DecisionTreeRegressor(ccp_alpha=143722.94076639024,random_state=1)
tree.fit(X, y)
pred = tree.predict(Xtest)
np.sqrt(mean_squared_error(test.price, pred))
```

```
7306.592294294368
```

The RMSE for the decision tree with cost complexity pruning is lower than that of linear regression models and spline regression models (including MARS), with these four predictors. However, it is higher than the one obtained with tuning tree parameters using grid search (shown previously). Cost complexity pruning considers a completely unpruned tree unlike the 'grid search' method, and thus may seem to be more comprehensive than the 'grid search' approach. However, the 'grid search' approach consideres several trees unlike cost complexity pruning that considers only one tree and prunes it. Thus, both approaches have advatages over each other, and either one may provide a more accurate model.

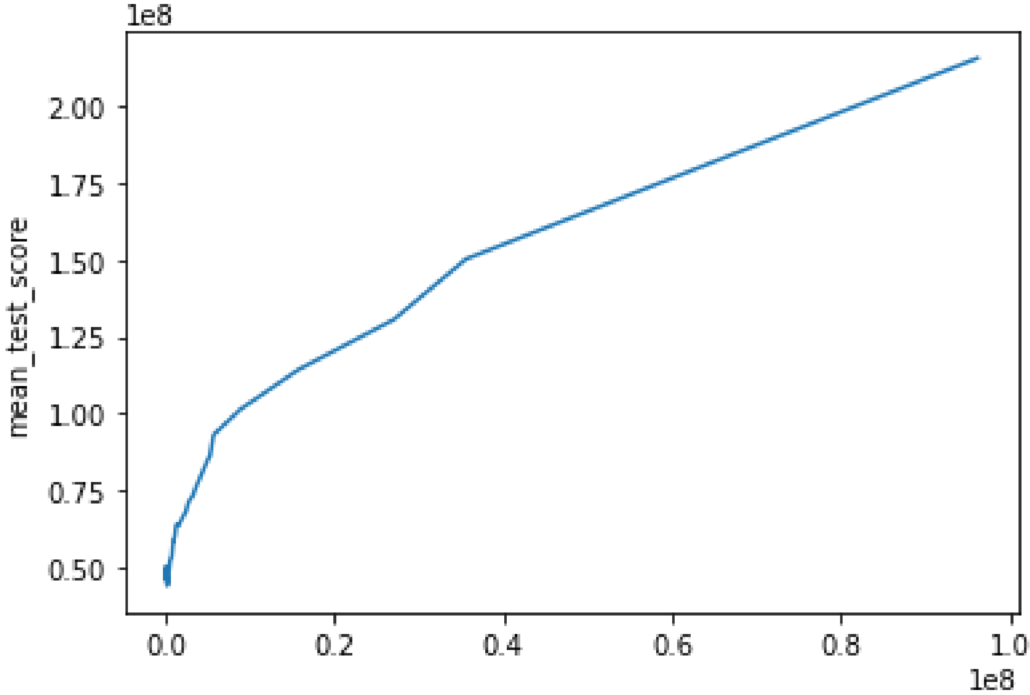
```python
gridcv_results = pd.DataFrame(model.cv_results_)
```

```python
cv_error = -gridcv_results['mean_test_score']
```

```python
#Vizualizing the 5-fold cross validation error vs alpha
sns.lineplot(alphas,cv_error)
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass t
  warnings.warn(
```

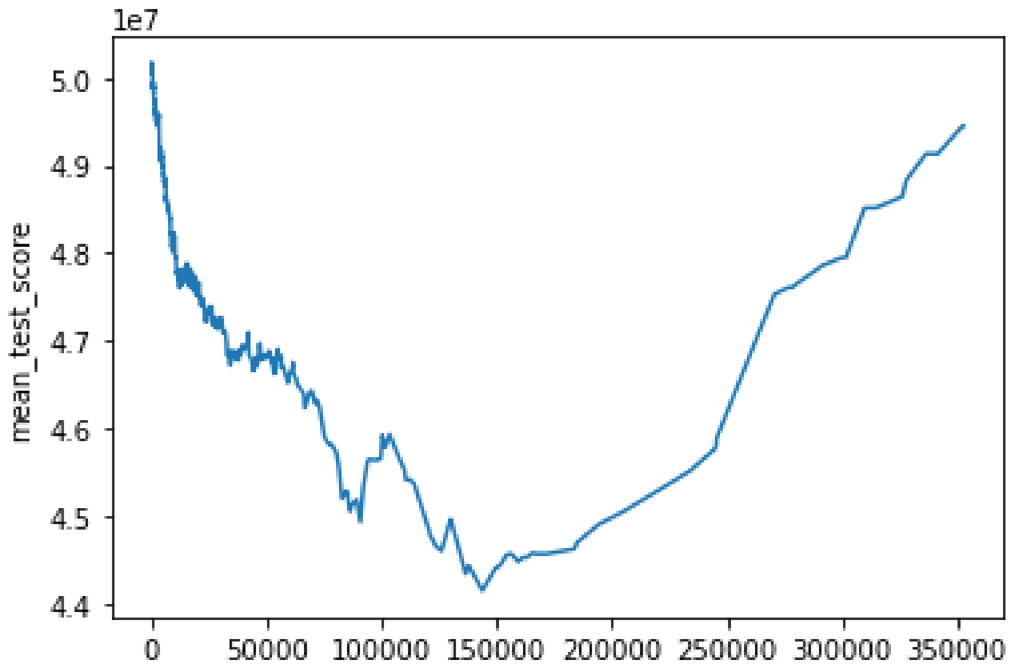
```
<AxesSubplot:ylabel='mean_test_score'>
```




```
#Zooming in the above vizualization to see the alpha where the 5-fold cross validation err
sns.lineplot(alphas[0:4093],cv_error[0:4093])
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass t
  warnings.warn(
```

```
<AxesSubplot:ylabel='mean_test_score'>
```



3.3.1 Depth vs alpha; Node counts vs alpha

```
stime = time.time()
trees=[]
for i in alphas:
    tree = DecisionTreeRegressor(ccp_alpha=i,random_state=1)
    tree.fit(X, train['price'])
    trees.append(tree)
print(time.time()-stime)
```

```
268.10325384140015
```

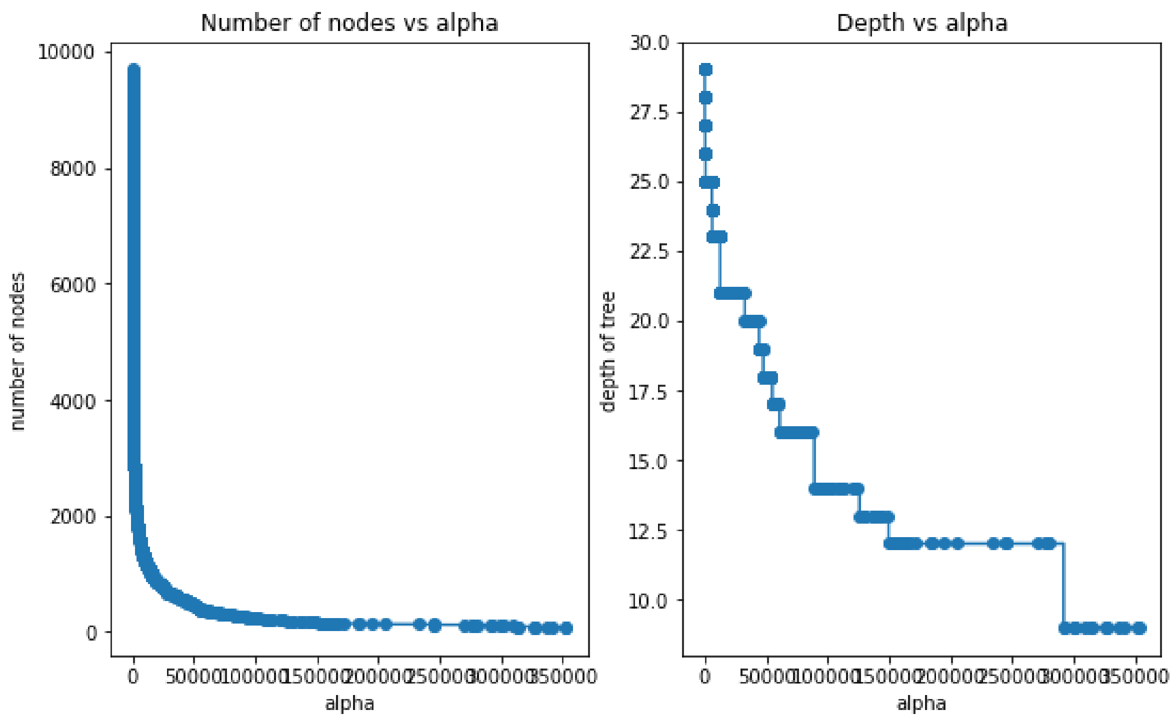
This code takes 4.5 minutes to run

```
node_counts = [clf.tree_.node_count for clf in trees]
depth = [clf.tree_.max_depth for clf in trees]
```

```
fig, ax = plt.subplots(1, 2,figsize=(10,6))
ax[0].plot(alphas[0:4093], node_counts[0:4093], marker="o", drawstyle="steps-post")#Plotti
```

```
ax[0].set_xlabel("alpha")
ax[0].set_ylabel("number of nodes")
ax[0].set_title("Number of nodes vs alpha")
ax[1].plot(alphas[0:4093], depth[0:4093], marker="o", drawstyle="steps-post")#Plotting the
ax[1].set_xlabel("alpha")
ax[1].set_ylabel("depth of tree")
ax[1].set_title("Depth vs alpha")
#fig.tight_layout()
```

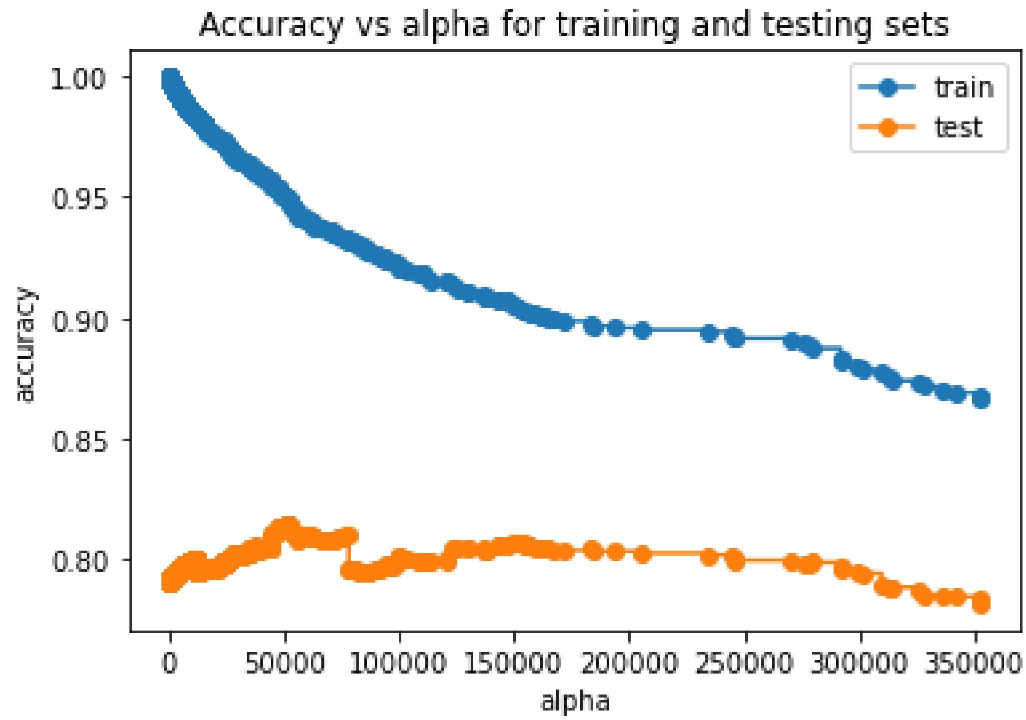
```
Text(0.5, 1.0, 'Depth vs alpha')
```



3.3.2 Train and test accuracies (R-squared) vs alpha

```
train_scores = [clf.score(X, y) for clf in trees]
test_scores = [clf.score(Xtest, test.price) for clf in trees]
```

```
fig, ax = plt.subplots()
ax.set_xlabel("alpha")
ax.set_ylabel("accuracy")
ax.set_title("Accuracy vs alpha for training and testing sets")
ax.plot(alphas[0:4093], train_scores[0:4093], marker="o", label="train", drawstyle="steps-
ax.plot(alphas[0:4093], test_scores[0:4093], marker="o", label="test", drawstyle="steps-po
ax.legend()
plt.show()
```



A Assignment A

1. You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and not as a group activity.
2. Do not write your name on the assignment.
3. Write your code in the *Code* cells and your answer in the *Markdown* cells of the Jupyter notebook. Ensure that the solution is written neatly enough to understand and grade.
4. Use Quarto to print the *.ipynb* file as HTML. You will need to open the command prompt, navigate to the directory containing the file, and use the command: `quarto render filename.ipynb --to html`. Submit the HTML file.
5. The assignment is worth 100 points, and is due on **Thursday, 13th April 2023 at 11:59 pm**.
6. **Four points are properly formatting the assignment**. The breakdown is as follows:

- Must be an HTML file rendered using Quarto (1 pt). *If you have a Quarto issue, you must mention the issue & quote the error you get when rendering using Quarto in the comments section of Canvas, and submit the ipynb file. If your issue doesn't seem genuine, you will lose points.*
- There aren't excessively long outputs of extraneous information (e.g. no printouts of entire data frames without good reason, there aren't long printouts of which iteration a loop is on, there aren't long sections of commented-out code, etc.) (1 pt)
- Final answers of each question are written in Markdown cells (1 pt).
- There is no piece of unnecessary / redundant code, and no unnecessary / redundant text (1 pt)

A.1 Bias-variance trade-off

Throughout the course, the conceptual clarity about bias and variance will help you tune the models for optimal performance and enable you to compare different models in terms of bias and variance. In this question, you will perform simulations to understand and visualize bias-variance trade-off as in Fig. 2.12 of the book (page 36).

Assume that the response y is a function of the predictors x_1 and x_2 and includes a random error ϵ, as follows:

$$y = f(x_1, x_2) + \epsilon, \tag{A.1}$$

where the function $f(.)$ is the Bukin function, $x_1 \sim U[-15, -5], x_2 \sim U[-3, 3]$, and $\epsilon \sim N(0, \sigma^2); \sigma = 10$. *Here U refers to Uniform distribution, and N refers to normal distribution.* Use NumPy to simulate values from these distributions.

You will code an algorithm (described below) to compute the expected squared bias, expected variance, var(ϵ) and expected test MSE of the following 7 linear regression models having the predictors as:

1. x_1 and x_2
2. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 2, which will be x_1^2, x_2^2, and x_1x_2
3. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 3, which will be $x_1^3, x_2^3, x_1^2x_2$, and $x_1x_2^2$
4. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 4
5. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 5
6. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 6
7. All the predictors in the above model, and all polynomial combinations of x_1, and x_2 of degree 7

As you can see the models are arranged in increasing order of flexibility / complexity. This corresponds to the horizontal axis of Fig. 2.12 in the book.

Use the following **algorithm** to compute the expected squared bias, expected variance, var(ϵ) and expected test MSE of the 7 linear regression models above:

I. Define the Bukin function that accepts x_1 and x_2 as parameters and returns the Bukin function value ($f(x_1, x_2)$).

(2 points)

II. Repeat steps **III - VII** for all degrees d in $\{1, 2, ..., 7\}$

(2 points)

III. Considering a model of **degree** d, simulate the following test and train datasets.

A. **Simulate test data**

1. Set a seed of 100. Use the code: `np.random.seed(100)`, where `np` refers to the numpy library
2. Simulate 100 values of x_1 from $U[-15, -5]$.
3. Simulate 100 values of x_2 from $U[-3, 3]$.
4. Compute the Bukin function value $f(x_1, x_2)$ for the simulated values of x_1 and x_2.
5. Use the function `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of x_1, and x_2 up to degree d.

(4 points)

B. **Simulate 100 train data sets**, where each train data is simulated as follows:

1. Set a seed of i for simualting the *ith* train data. Use the code: `np.random.seed(i)`, where `np` refers to the numpy library.
2. Simulate 100 values of x_1 from $U[-15, -5]$
3. Simulate 100 values of x_2 from $U[-3, 3]$
4. Compute the Bukin function value $f(x_1, x_2)$ for the simulated values of x_1 and x_2
5. Simulate the response y using the above set of simulated values with Equation A.1
6. Use the function `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of x_1, and x_2 up to degree d.

(6 points)

IV. For each train data in III(B), develop a linear regression model using the `LinearRegression()` function from the `linear_model` module of the `sklearn` library.

(2 points)

V. Note that the squared bias at a test point x_{1_test}, x_{2_test} is:

$$[Bias(\hat{f}(x_{1\_test}, x_{2\_test}))]^2 = [E(\hat{f}(x_{1\_test}, x_{2\_test})) - f(x_{1\_test}, x_{2\_test})]^2, \tag{A.2}$$

where $E(\hat{f}(x_{1\_test}, x_{2\_test}))$ is the mean prediction of the 100 trained models at x_{1_test}, x_{2_test}.

Compute the overall expected squared bias as the average squared bias at all the test data points, as in the equation below:

$$[Bias(\hat{f}(.))]^2 = \frac{1}{100}\Sigma_{i=1}^{100}\left[Bias(\hat{f}(x_{1i\_test}, x_{2i\_test}))\right]^2, \tag{A.3}$$

(8 points)

VI. Note that the variance at a test point x_{1_test}, x_{2_test} is $Var(\hat{f}(x_{1\_test}, x_{2\_test}))$. Compute the overall expected variance as the average variance at all the test data points, as in the equation below:

$$Var(\hat{f}(.)) = \frac{1}{100}\Sigma_{i=1}^{100} Var(\hat{f}(x_{1i\_test}, x_{2i\_test})) \tag{A.4}$$

(6 points)

VII. Compute the overall expected test mean squared error as the sum of the expected squared bias (Equation A.3), expected variance (Equation A.4), and error variance (σ^2):

$$MSE = [Bias(\hat{f}(.))]^2 + Var(\hat{f}(.)) + \sigma^2, \tag{A.5}$$

(4 points)

VIII. Plot the overall expected squared bias, overall expected variance, and overall expected test MSE *(as obtained from Equation A.3, Equation A.4, and Equation A.5 respectively)* against the degree d (or flexibility / complexity) of the model . Your plot should look like one of the plots in Fig. 2.12 of the book.

(3 points)

IX. What is the degree of the optimal model, i.e., the degree that provides the best **bias-variance trade-off**?

(2 points)

Note: While coding the algorithm, comment it well so that it is easy to give partial credit in case of mistakes. Include the numerals of the algorithm (such as II(B), V, VI, etc.) in your comments so that it is easy to check your algorithm for completeness.

A.2 Tuning a classification model with `sklearn`

Data

Read the data *classification_data.csv*. The description of the columns is as follows:

1. `hi_int_prncp_pd`: Indicates if a high percentage of the repayments made went to interest rather than principal. **Target variable.**
2. `out_prncp_inv`: Remaining outstanding principal for portion of total amount funded by investors
3. `loan_amnt`: The listed amount of the loan applied for by the borrower. If at some point in time, the credit department reduces the loan amount, then it will be reflected in this value.
4. `int_rate`: Interest rate on the loan
5. `term`: The number of payments on the loan. Values are in months and can be either 36 or 60.

You will develop and tune a logistic regression model to predict `hi_int_prncp_pd` based on the rest of the columns (predictors) as per the instructions below.

A.2.1 Train-test split

Use the function `train_test_split` from the `model_selection` module of the `sklearn` library to split the data into 75% train and 25% test. Stratify the split based on the response. Use `random_state` as 45. Print the proportion of 0s and 1s in both the train and test datasets.

(4 points)

A.2.2 Scaling predictors

Scale the predictors to avoid convergence errors when fitting the logistic regression model.

Note that last quarter, we were focusing on inference (along with prediction), so we avoided scaling. It is a bit inconvenient to interpret odds with scaled predictors. However, avoiding scaling may lead to convergence errors as some of you saw in your course projects. So, it is a good practice to scale, especially when your focus is prediction.

(3 points)

A.2.3 Tuning the degree

Use the functions:

1. `cross_val_score` from the `model_selection` module of the `sklearn` library to tune the degree of the logistic regression model for maximizing the stratified 5-fold prediction accuracy. Consider degrees from 1 to 6.

2. `PolynomialFeatures` from the `preprocessing` module of the `sklearn` library to create all polynomial combinations of the predictors up to degree d.

What is the optimal degree?

(4 points)

Notes:

- A model of degree d will consist of polynomial transformations and interactions of predictors up to degree d. For example, a model of degree 2 will consist of the square of each predictor and all 2-factor interactions of the predictors.
- You may use the `newton-cg` solver to avoid convergence issues.
- Use the default `C` value at this point, you will tune it later.

A.2.4 Test accuracy with optimal degree

For the optimal degree identified in the previous question, compute the test accuracy.

(4 points)

A.2.5 Tuning `C`

With the optimal degree identified in the previous question, find the optimal regularization parameter `C`. Again use the `cross_val_score` function.

(3 points)

A.2.6 Test accuracy with optimal degree and `C`

For the optimal degree and optimal `C` identified in the previous questions, compute the test accuracy.

(3 points)

A.2.7 Tuning decision threshold probability

With the optimal degree and optimal `C` identified in the previous questions, find the optimal decision threshold probability to maximize accuracy. Use the `cross_val_predict` function.

(4 points)

A.2.8 Test accuracy for optimal degree, `C`, and threshold probability

For the optimal degree, optimal `C`, and optimal decision threshold probabilities identified in the previous questions, compute the test accuracy.

(4 points)

A.2.9 Simultaneous optimization of multiple parameters

In the above tuning approach we optimized the hyperparameters and the decision threshold probability sequentially. This is a greedy approach, which doesn't consider all combinations of hyperparameters and decision threshold probabilities, and thus may fail to find the optimal combination of values that maximize accuracy. Thus, tune both the model hyperparameters - degree and `C`, and the decision threshold probability simultaneously considering all value combinations. This will take more time, but is likely to provide more accurate optimal parameter values.

(6 points)

A.2.10 Test accuracy with optimal parameters obtained simultaneously

For the optimal degree, optimal `C`, and optimal decision threshold probabilities identified in the previous question, compute the test accuracy.

(4 points)

A.2.11 Optimizing parameters for multiple performance metrics

Find the optimal `C` and degree to maximize recall while having a precision of more than 75%. Use the function `cross_validate` from the `model_selection` module of the `sklearn` library.

Note: `cross_validate` function is very similar to `cross_val_score`, the only difference is you can use multiple metrics with the scoring input, as you need in this question.

(8 points)

A.2.12 Performance metrics computation

For the optimal degree and `C` identified in the previous question, compute the following performance metrics on test data. Use `sklearn` functions, manual computation is not allowed.

1. Precision
2. Recall
3. Accuracy
4. ROC-AUC
5. Show the confusion matrix

(10 points)

B Stratified splitting (classification problem)

B.1 Stratified splitting with respect to response

Q: When splitting data into train and test for developing and assessing a classification model, it is recommended to stratify the split with respect to the response. Why?

A: The main advantage of stratified splitting is that it can help ensure that the training and testing sets have similar distributions of the target variable, which can lead to more accurate and reliable model performance estimates.

In many real-world datasets, the target variable may be imbalanced, meaning that one class is more prevalent than the other(s). For example, in a medical dataset, the majority of patients may not have a particular disease, while only a small fraction may have the disease. If a random split is used to divide the dataset into training and testing sets, there is a risk that the testing set may not have enough samples from the minority class, which can lead to biased model performance estimates.

Stratified splitting addresses this issue by ensuring that both the training and testing sets have similar proportions of the target variable. This can lead to more accurate model performance estimates, especially for imbalanced datasets, by ensuring that the testing set contains enough samples from each class to make reliable predictions.

Another advantage of stratified splitting is that it can help ensure that the model is not overfitting to a particular class. If a random split is used and one class is overrepresented in the training set, the model may learn to predict that class well but perform poorly on the other class(es). Stratified splitting can help ensure that the model is exposed to a representative sample of all classes during training, which can improve its generalization performance on new, unseen data.

In summary, the advantages of stratified splitting are that it can lead to more accurate and reliable model performance estimates, especially for imbalanced datasets, and can help prevent overfitting to a particular class.

B.2 Stratified splitting with respect to response and categorical predictors

Q: Will it be better to stratify the split with respect to the response as well as categorical predictors, instead of only the response? In that case, the train and test datasets will be even more representative of the complete data.

A: It is not recommended to stratify with respect to both the response and categorical predictors simultaneously, while splitting a dataset into train and test, because doing so may result in the test data being very similar to train data, thereby defeating the purpose of assessing the model on unseen data. This kind of a stratified splitting will tend to make the relationships between the response and predictors in train data also appear in test data, which will result in the performance on test data being very similar to that in train data. Thus, in this case, the ability of the model to generalize to new, unseen data won't be assessed by test data.

Therefore, it is generally recommended to only stratify the response variable when splitting the data for model training, and to use random sampling for the predictor variables. This helps to ensure that the model is able to capture the underlying relationships between the predictor variables and the response variable, while still being able to generalize well to new, unseen data.

In the extreme scenario, when there are no continuous predictors, and there are enough observations for stratification with respect to the response and the categorical predictors, the train and test datasets may turn out to be exactly the same. Example 1 below illustrates this scenario.

B.3 Example 1

The example below shows that the train and test data can be exactly the same if we stratify the split with respect to response and the categorical predictors.

```
# Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_predict, cross_val_score
from sklearn.metrics import accuracy_score
from itertools import product
sns.set(font_scale=1.35)
```

Let us simulate a dataset with 8 observations, two categorical predictors `x1` and `x2` and the the binary response `y`.

```
#Setting a seed for reproducible results
np.random.seed(9)

# 8 observations
n = 8

#Simulating the categorical predictors
x1 = pd.Series(np.random.randint(0,2,n), name = 'x1')
x2 = pd.Series(np.random.randint(0,2,n), name = 'x2')

#Simulating the response
pr = (x1==1)*0.7+(x2==0)*0.3# + (x3*0.1>0.1)*0.1
y = pd.Series(1*(np.random.uniform(size = n) < pr), name = 'y')

#Defining the predictor object 'X'
X = pd.concat([x1, x2], axis = 1)

#Stratified splitting with respect to the response and predictors to create 50% train and
X_train_stratified, X_test_stratified, y_train_stratified,\
y_test_stratified = train_test_split(X, y, test_size = 0.5, random_state = 45, stratify=da

#Train and test data resulting from the above stratified splitting
data_train = pd.concat([X_train_stratified, y_train_stratified], axis = 1)
data_test = pd.concat([X_test_stratified, y_test_stratified], axis = 1)
```

Let us check the train and test datasets created with stratified splitting with respect to both the predictors and the response.

```
data_train
```

	x1	x2	y
2	0	0	1
7	0	1	0
3	1	0	1
1	0	1	0

```
data_test
```

	x1	x2	y
4	0	1	0
6	1	0	1
0	0	1	0
5	0	0	1

Note that the train and test datasets are exactly the same! Stratified splitting tends to have the same proportion of observations corresponding to each strata in both the train and test datasets, where each strata is a unique combination of values of `x1`, `x2`, and `y`. This will tend to make the train and test datasets quite similar!

B.4 Example 2: Simulation results

The example below shows that train and test set performance will tend to be quite similar if we stratify the datasets with respect to the predictors and the response.

We'll simulate a dataset consisting of 1000 observations, 2 categorical predictors `x1` and `x2`, a continuous predictor `x3`, and a binary response `y`.

```
#Setting a seed for reproducible results
np.random.seed(99)

# 1000 Observations
n = 1000

#Simulating categorical predictors x1 and x2
x1 = pd.Series(np.random.randint(0,2,n), name = 'x1')
x2 = pd.Series(np.random.randint(0,2,n), name = 'x2')

#Simulating continuous predictor x3
x3 = pd.Series(np.random.normal(0,1,n), name = 'x3')

#Simulating the response
pr = (x1==1)*0.7+(x2==0)*0.3 + (x3*0.1>0.1)*0.1
y = pd.Series(1*(np.random.uniform(size = n) < pr), name = 'y')

#Defining the predictor object 'X'
X = pd.concat([x1, x2, x3], axis = 1)
```

We'll comparing model performance metrics when the data is split into train and test by performing stratified splitting

1. Only with respect to the response
2. With respect to the response and categorical predictors

We'll perform 1000 simulations, where the data is split using a different seed in each simulation.

```
#Creating an empty dataframe to store simulation results of 1000 simulations
accuracy_iter = pd.DataFrame(columns = {'train_y_stratified','test_y_stratified',
                                        'train_y_CatPredictors_stratified','test_y_CatPred
```

```
# Comparing model performance metrics when the data is split into train and test by perfor
# (1) only with respect to the response
# (2) with respect to the response and categorical predictors

# Stratified splitting is performed 1000 times and the results are compared
for i in np.arange(1,1000):

    #--------Case 1-------------------#
    # Stratified splitting with respect to response only to create train and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_stat
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Model accuracy on train and test data, with stratification only on response while sp
    # the complete data into train and test
    accuracy_iter.loc[(i-1), 'train_y_stratified'] = model.score(X_train, y_train)
    accuracy_iter.loc[(i-1), 'test_y_stratified'] = model.score(X_test, y_test)

    #--------Case 2-------------------#
    # Stratified splitting with respect to response and categorical predictors to create t
    # and test data
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_stat
                                                        stratify=pd.concat([x1, x2, y], ax
    model.fit(X_train, y_train)

    # Model accuracy on train and test data, with stratification on response and predictor
    # splitting the complete data into train and test
    accuracy_iter.loc[(i-1), 'train_y_CatPredictors_stratified'] = model.score(X_train, y_
    accuracy_iter.loc[(i-1), 'test_y_CatPredictors_stratified'] = model.score(X_test, y_te

# Converting accuracy to numeric
```

```
accuracy_iter = accuracy_iter.apply(lambda x:x.astype(float), axis = 1)
```

Distribution of train and test accuracies

The table below shows the distribution of train and test accuracies when the data is split into train and test by performing stratified splitting:

1. Only with respect to the response (see `train_y_stratified` and `test_y_stratified`)
2. With respect to the response and categorical predictors (see `train_y_CatPredictors_stratified` and `test_y_CatPredictors_stratified`)

```
accuracy_iter.describe()
```

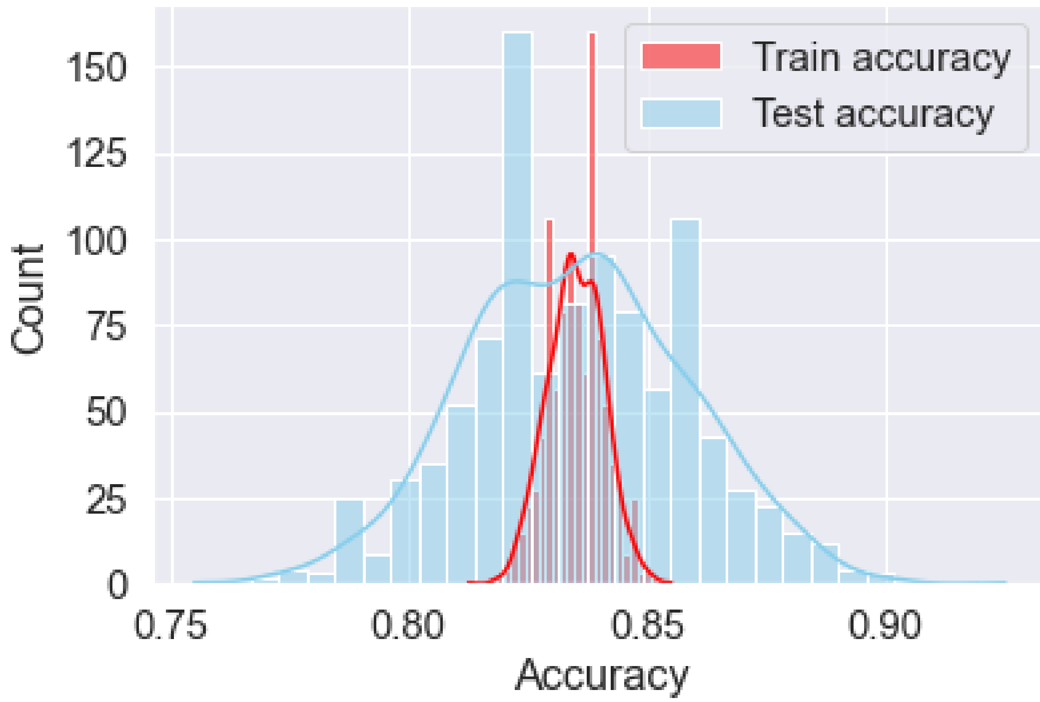
	train_y_stratified	test_y_stratified	train_y_CatPredictors_stratified	test_y_CatPredictors_st
count	999.000000	999.000000	9.990000e+02	9.990000e+02
mean	0.834962	0.835150	8.350000e-01	8.350000e-01
std	0.005833	0.023333	8.552999e-15	8.552999e-15
min	0.812500	0.755000	8.350000e-01	8.350000e-01
25%	0.831250	0.820000	8.350000e-01	8.350000e-01
50%	0.835000	0.835000	8.350000e-01	8.350000e-01
75%	0.838750	0.850000	8.350000e-01	8.350000e-01
max	0.855000	0.925000	8.350000e-01	8.350000e-01

Let us visualize the distribution of these accuracies.

B.4.1 Stratified splitting only with respect to the response

```
sns.histplot(data=accuracy_iter, x="train_y_stratified", color="red", label="Train accurac
sns.histplot(data=accuracy_iter, x="test_y_stratified", color="skyblue", label="Test accur
plt.legend()
plt.xlabel('Accuracy')
```

```
Text(0.5, 0, 'Accuracy')
```

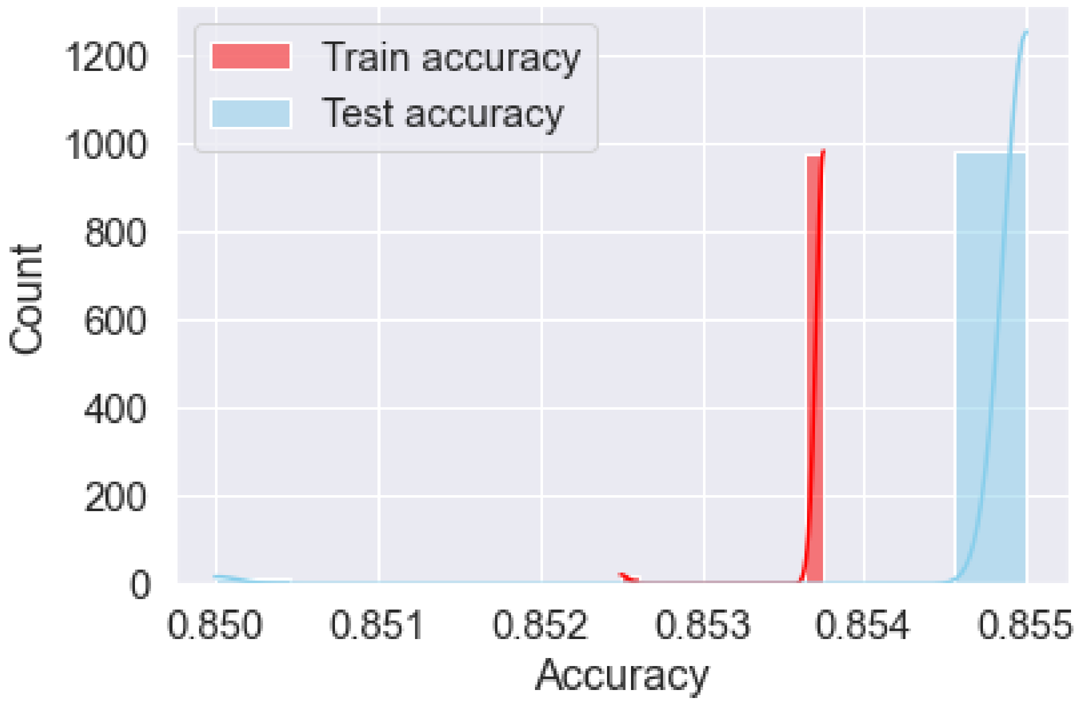


Note the variability in train and test accuracies when the data is stratified only with respect to the response. The train accuracy varies between 81.2% and 85.5%, while the test accuracy varies between 75.5% and 92.5%.

B.4.2 Stratified splitting with respect to the response and categorical predictors

```
sns.histplot(data=accuracy_iter, x="train_y_CatPredictors_stratified", color="red", label=
sns.histplot(data=accuracy_iter, x="test_y_CatPredictors_stratified", color="skyblue", lab
plt.legend()
plt.xlabel('Accuracy')
```

```
Text(0.5, 0, 'Accuracy')
```



The train and test accuracies are between 85% and 85.5% for all the simulations. As a results of stratifying the splitting with respect to both the response and the categorical predictors, the train and test datasets are almost the same because the datasets are engineered to be quite similar, thereby making the test dataset inappropriate for assessing accuracy on unseen data. Thus, it is recommended to stratify the splitting only with respect to the response.

C Assignment A

This notebook shows an example on how to tune the value of a hyperparameter. The example considered is Question A.2.5 of Assignment A, where we need to tune the regularization parameter `C` for a logistic regression model. With this example, you should understand:

1. How to think about the range of values to consider for tuning a hyperparameter.
2. How should the values under consideration be distributed in the range identified in (1).

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_predict, cross_val_score,
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, roc_curve, auc, precision_score, recall_score,
```

```
data = pd.read_csv('./Datasets/classification_data.csv')

X = data.drop(columns= 'hi_int_prncp_pd')
y = data['hi_int_prncp_pd']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state =
```

```
scaler = StandardScaler().fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

With the optimal degree identified in the previous question, find the optimal regularization parameter `C`. Again use the `cross_val_score` function.

(4 points)

We are tuning `C` for the optimal degree of 5 identified in one of the previous questions.

```
poly = PolynomialFeatures(degree = 5)
X_train_poly = poly.fit_transform(X_train_scaled)
```

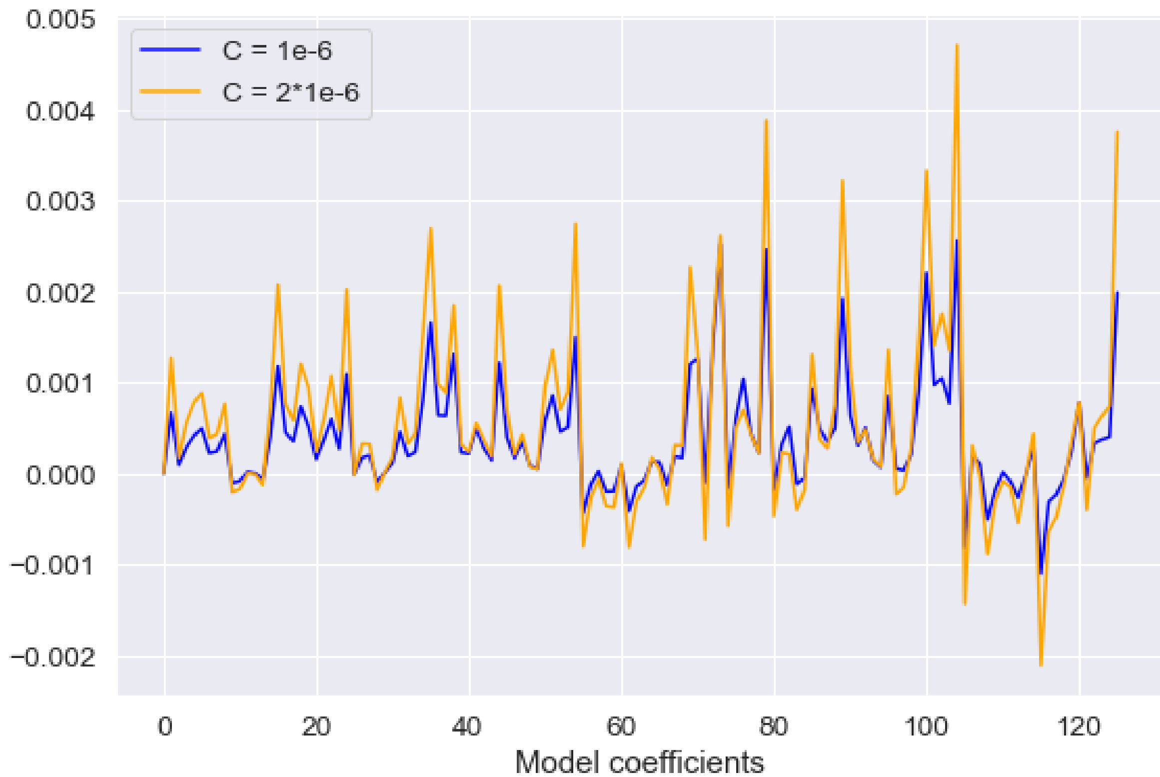
C.1 What should be the minimum value of C to consider?

- As C is the regularization parameter, it cannot be negative.
- As C = 1/lambda, a value of C = 0 will mean infinite regularization, which corresponds to an intercept-only model. Also, as the the `LogisticRegression()` function computes the value of lambda as `1/C`, it throws a *division by zero error* if `C = 0`. Thus, we should consider `C>0`.
- Even if C is positive, but very small, the value of lambda will be too high, which gives rise to numerical errors. Thus, we need to find the minimum value of C that is large enough to avoid numerical errors when fitting the model to the given standardized dataset.

We start with an extremely low value of `C = 1e-10`, and check if the model converges. It doesn't converge! If it had converged, we will consider even lower values of C. However, in this case, it fails to converge indicating the possibility that this value of C is too small to avoid numerical errors. We keep increasing the order of C until we don't see convergence errors. With the code below, we find that for values of C starting from `1e-6`, the algorithm successfully converges. Thus, the minimum value of C that we consider will be `1e-6`.

In the plot below, we also see that as we increase C starting from `C = 1e-6`, the model coefficients change, which may potentially change model fit and accuracy. Thus, we should consider increasing values of C starting from `C = 1e-6`.

```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e-6, max_iter=100).fit(X_train_poly,
model2 = LogisticRegression(solver = 'newton-cg', C = 2*1e-6).fit(X_train_poly, y_train)

# Visualizing the model coefficients with changing values of 'C'
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e-6")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 2*1e-6");
plt.xlabel('Model coefficients')
plt.legend();
```



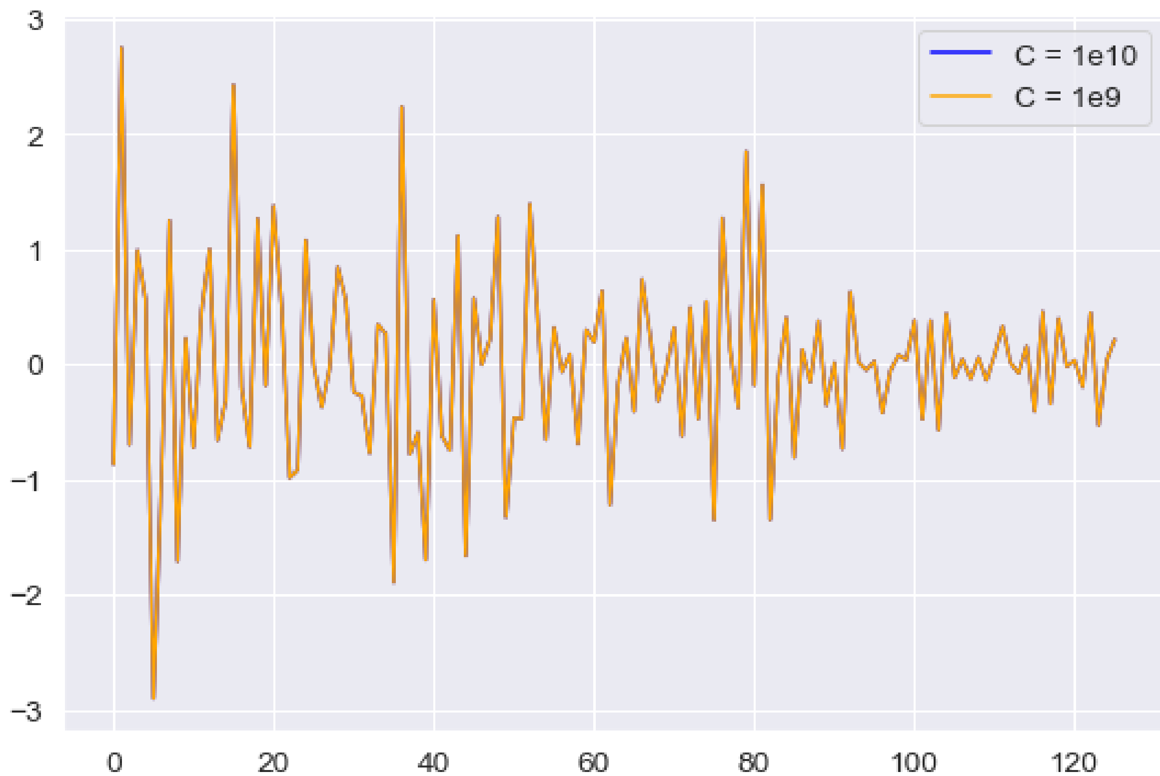
C.2 What should be the maximum value of `C` to consider?

As `C` tends to infinity, the regularization tends to disappear. Let us consider values of `C` starting from `C = 1e10`. The algorithm converges, and we obtain a plot as shown below.

However, do we need to start from values as high as `1e10`?

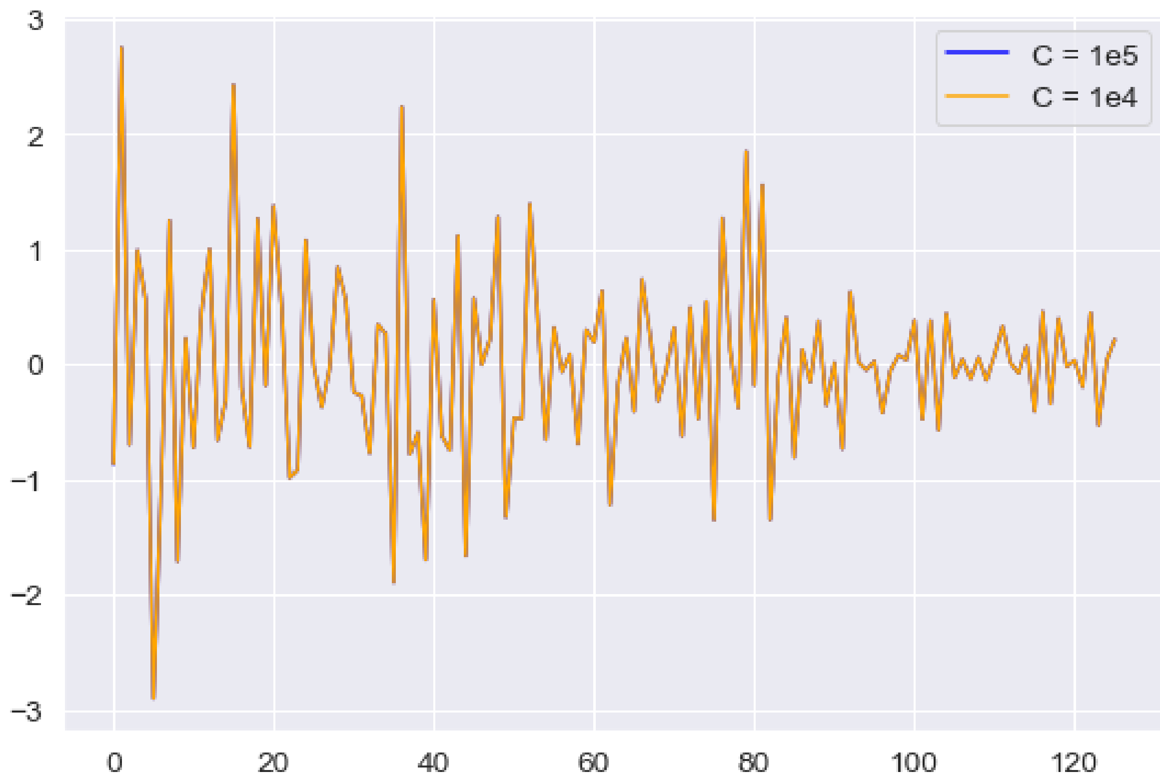
No, if we check the coefficients for `C = 1e9`, they appear to be the same as the coefficients for `C = 1e10` (see plot below). Thus, we need to identify the maximum value of `C` below which the coefficients tend to change when the value of `C` decreases further.

```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e10).fit(X_train_poly, y_train)
model2 = LogisticRegression(solver = 'newton-cg', C = 1e9).fit(X_train_poly, y_train)
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e10")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e9")
plt.legend();
```



There doesn't seem to be a difference even between `C = 1e5` and `C = 1e4` - both the values are still practically infinity. Let us reduce `C` further.

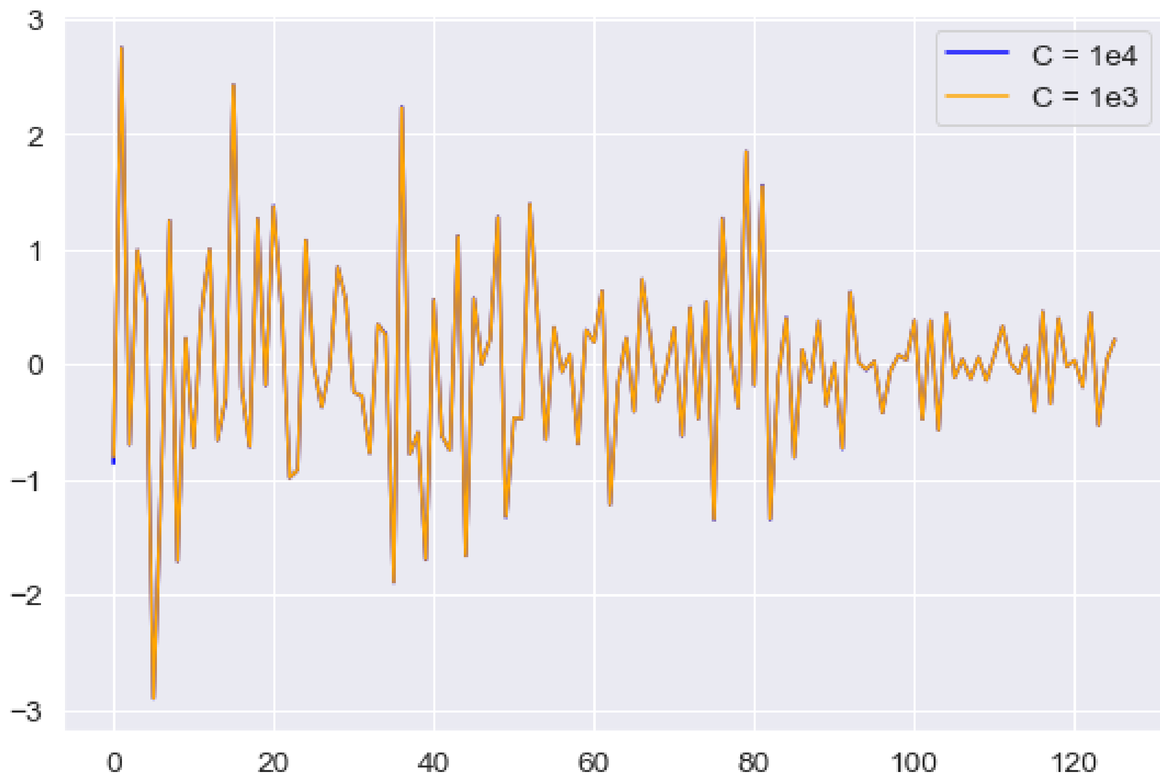
```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e5).fit(X_train_poly, y_train)
model2 = LogisticRegression(solver = 'newton-cg', C = 1e4).fit(X_train_poly, y_train)
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e5")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e4")
plt.legend();
```



Let us consider `C = 1e3`. We get a converence error. As the solution is found due to algorithms such as gradient descent, the algorithm may just need more steps or more iterations to converge to a solution. Thus, we can try increasing the `max_iter` value to see if it helps the model converge.

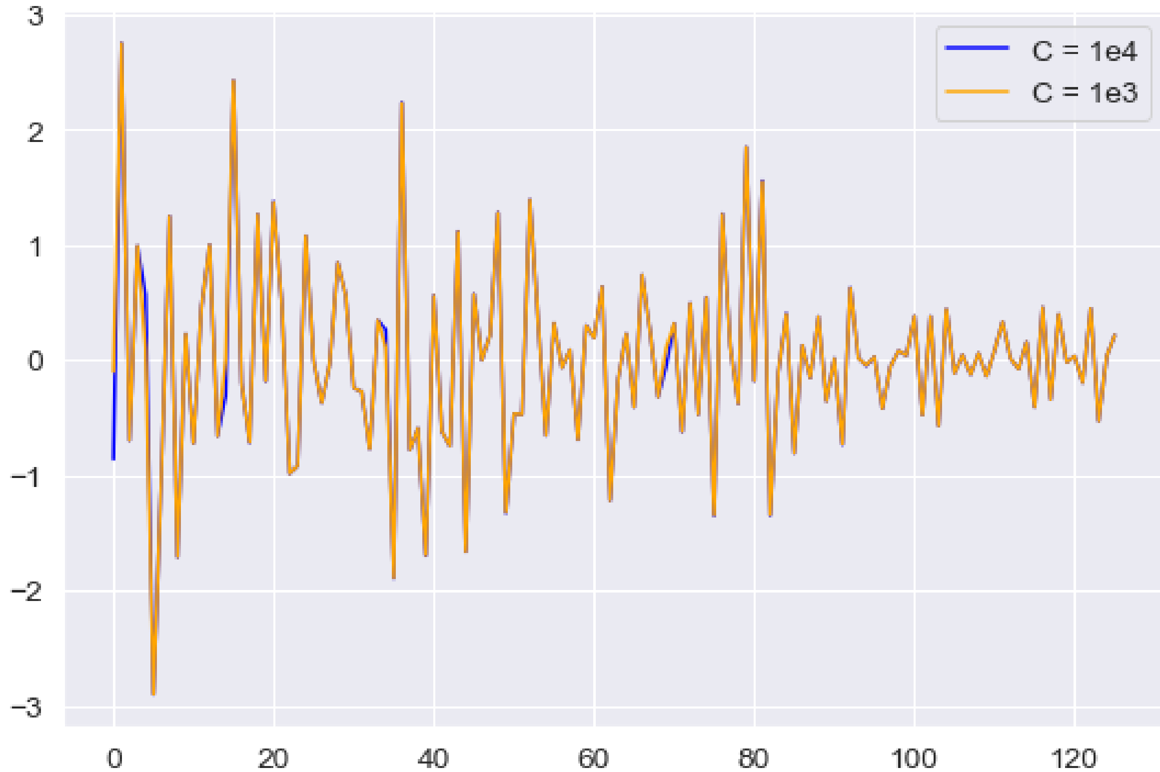
```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e4).fit(X_train_poly, y_train)
model2 = LogisticRegression(solver = 'newton-cg', C = 1e3).fit(X_train_poly, y_train)
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e4")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e3")
plt.legend();
```

```
C:\Users\akl0407\Anaconda3\lib\site-packages\sklearn\utils\optimize.py:202: ConvergenceWarni
  warnings.warn("newton-cg failed to converge. Increase the "
```



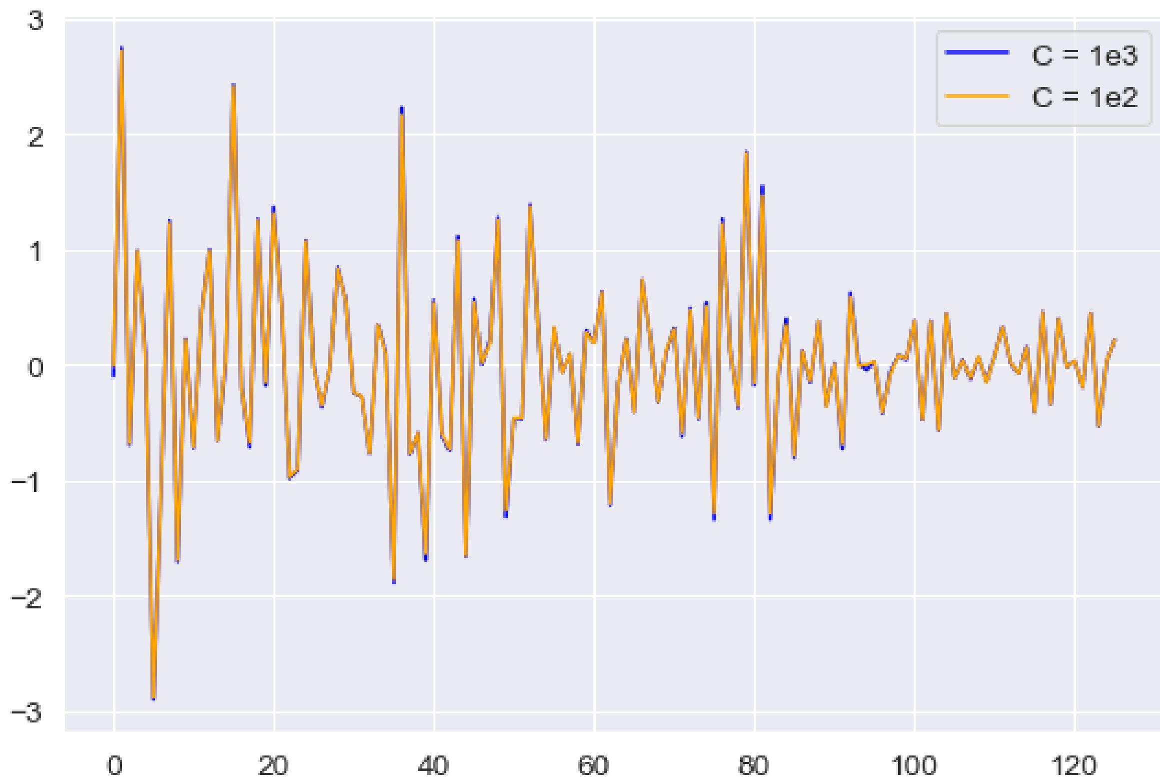
Increasing the `max_iter` value does take more time for the code to execute, but it helps the algorithm converge to a solution (see below). However, we see that the coefficients are very similar for the two values of `C`. Thus, we can decrease `C` further.

```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e4).fit(X_train_poly, y_train)
model2 = LogisticRegression(solver = 'newton-cg', C = 1e3, max_iter=1000).fit(X_train_poly
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e4")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e3")
plt.legend();
```



Even for `C = 1e3` and `C = 1e2`, we have similar coefficients. Let us reduce `C` further.

```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e3, max_iter=1000).fit(X_train_poly,
model2 = LogisticRegression(solver = 'newton-cg', C = 1e2, max_iter=1000).fit(X_train_poly
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e3")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e2")
plt.legend();
```



As we decrease `C` from `C = 1e2`, we observe that the coefficients start changing. Thus, the maximum value of `C` that we should consider is `C = 1e2`, as this value is practically infinity, and higher values will not be useful for consideration.

```
sns.set(font_scale=1.25)
plt.rcParams["figure.figsize"] = (9,6)
model = LogisticRegression(solver = 'newton-cg', C = 1e2, max_iter=1000).fit(X_train_poly,
model2 = LogisticRegression(solver = 'newton-cg', C = 1e1, max_iter=1000).fit(X_train_poly
plt.plot(range(126), model.coef_[0,:], color = 'blue', label = "C = 1e2")
plt.plot(range(126), model2.coef_[0,:], color = 'orange', label = "C = 1e1")
plt.legend();
```

C.3 Grid search: Coarse grid

Let us consider 50 values of `C` between the minimum and maximum values identified above. We'll consider values of `C` equidistant in the log scale, so that we consider values of all orders (such as `1e-5`, `1e-4`, etc.). Also, we saw earlier that the model coefficients change as the order of values of `C` changes from `1e-6` to `1e-5`. Thus, we should consider values of `C` equidistant in logscale, instead of the linear scale.

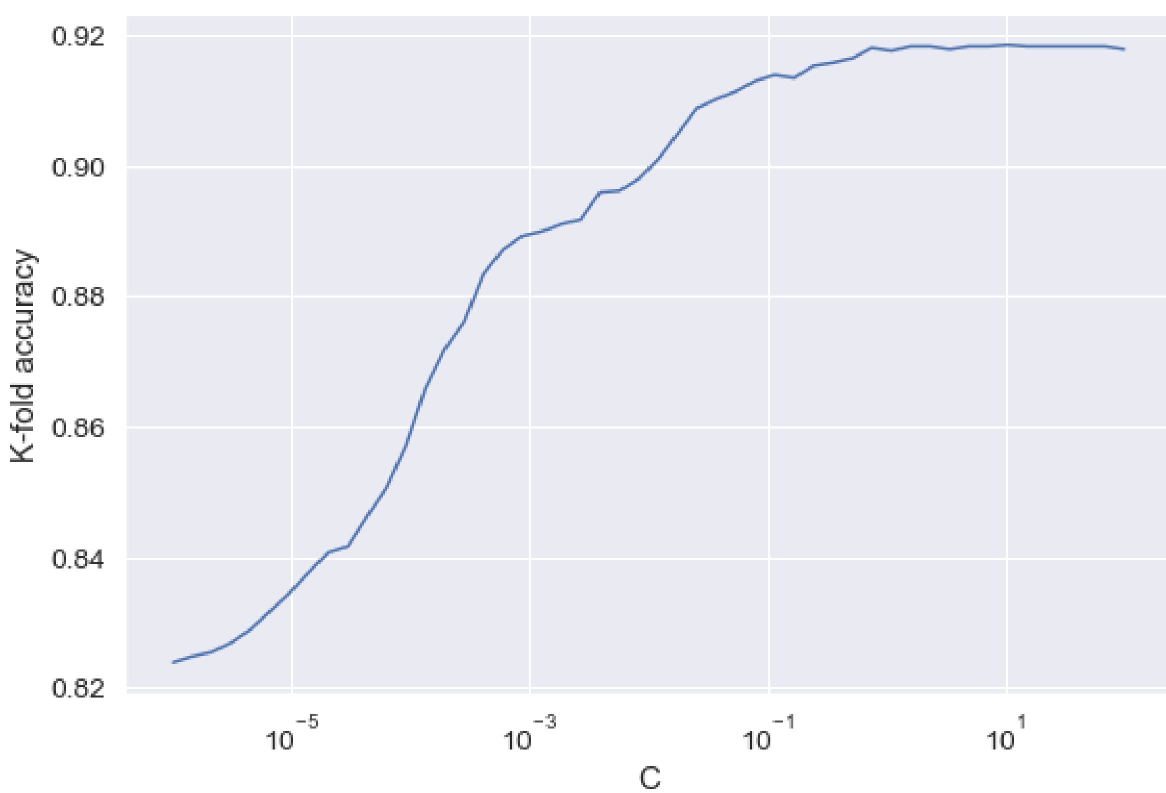
```
hyperparam_vals = np.logspace(-6,2)
accuracy_iter = []
for c_val in hyperparam_vals:
    print(c_val)
    poly = PolynomialFeatures(degree = 5)
    X_train_poly = poly.fit_transform(X_train_scaled)
    accuracy_iter.append(cross_val_score(LogisticRegression(solver = 'newton-cg', C = c_va
                                         X_train_poly,
                                                  y_train, cv = 5, scoring='accuracy'))
```

```
1e-06
1.4563484775012444e-06
2.1209508879201924e-06
3.0888435964774785e-06
4.498432668969444e-06
6.5512855685955095e-06
9.540954763499944e-06
1.3894954943731361e-05
2.0235896477251556e-05
2.94705170255181e-05
4.291934260128778e-05
6.250551925273976e-05
9.102981779915228e-05
0.00013257113655901082
0.00019306977288832496
0.0002811768697974228
0.0004094915062380423
0.0005963623316594642
0.000868511373751352
0.0012648552168552957
0.0018420699693267144
0.0026826957952797246
0.003906939937054613
0.005689866029018293
0.008286427728546842
0.012067926406393288
0.017575106248547894
0.025595479226995333
0.03727593720314938
0.054286754393238594
0.07906043210907686
0.11513953993264457
0.16768329368110066
0.244205309454865
0.35564803062231287
0.5179474679231202
0.7543120063354607
1.0985411419875573
1.5998587196060574
2.329951810515367
3.393221771895323
4.941713361323828
7.196856730011514
```

```
10.481131341546853
15.264179671752302
22.22996482526191
32.3745754281764
47.1486636345739
68.66488450042998
100.0
```

Next, we plot the 5-fold accuracy with increasing `C`.

```python
#K-fold accuracy vs C
acc_vector = np.array(accuracy_iter).mean(axis=1)
plt.plot(10**np.linspace(-6, 2), acc_vector)
plt.xscale("log")
plt.xlabel('C')
plt.ylabel('K-fold accuracy');
```



```
hyperparam_vals[np.argmax(np.array(accuracy_iter).mean(axis=1))]
```

```
10.481131341546853
```

We observe that the accuracy is the maximum when `C` is more than 0.1. Thus, we'll zoom-in and search for the optimal value in the domain `0.1 < C < 100` to obtain a more precise estimate of optimal `C`.

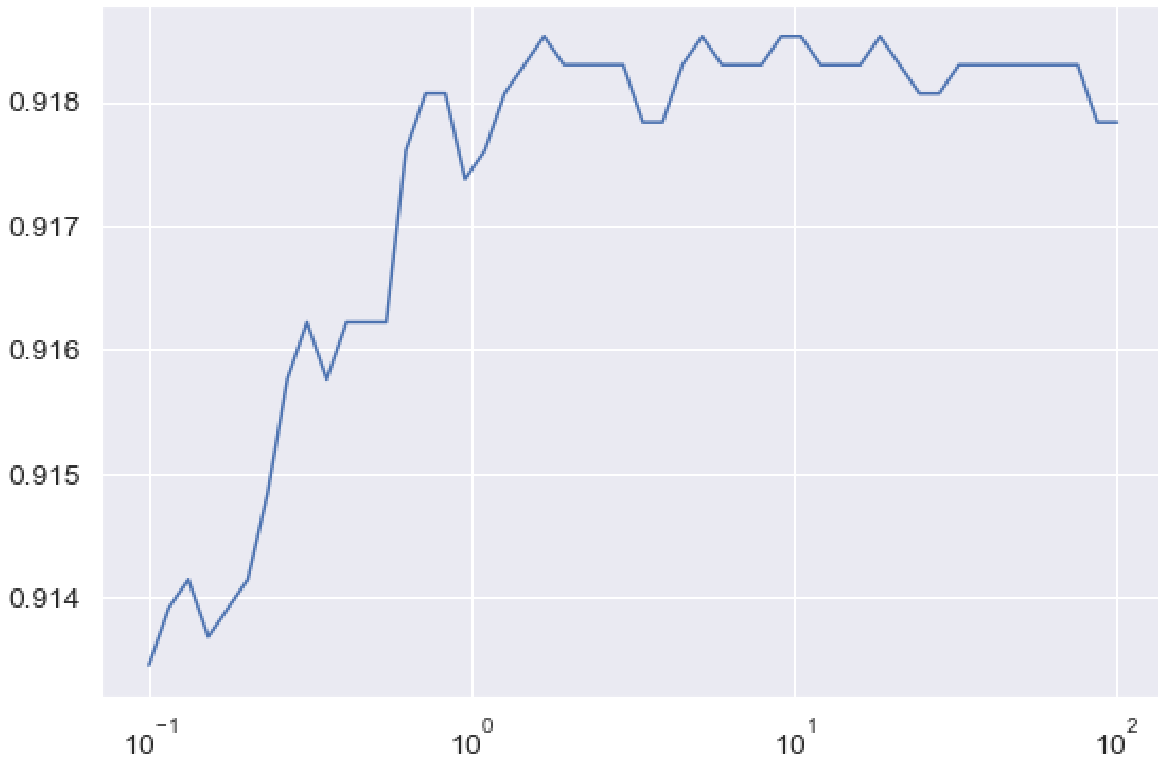
C.4 Grid search: Finer grid

```
hyperparam_vals = np.logspace(-1,2)
accuracy_iter2 = []
for c_val in hyperparam_vals:
    print(c_val)
    poly = PolynomialFeatures(degree = 5)
    X_train_poly = poly.fit_transform(X_train_scaled)
    accuracy_iter2.append(cross_val_score(LogisticRegression(solver = 'newton-cg', C = c_v
                                          X_train_poly,
                                                   y_train, cv = 5, scoring='accuracy'))
```

```
0.1
0.11513953993264472
0.13257113655901093
0.15264179671752334
0.1757510624854792
0.20235896477251572
0.2329951810515372
0.2682695795279726
0.3088843596477481
0.35564803062231287
0.40949150623804254
0.47148663634573934
0.5428675439323859
0.6250551925273973
0.7196856730011519
0.8286427728546845
0.9540954763499939
1.0985411419875584
```

```
1.2648552168552958
1.4563484775012436
1.6768329368110082
1.9306977288832496
2.2229964825261943
2.559547922699557
2.9470517025518097
3.393221771895328
3.906939937054617
4.498432668969446
5.17947467923121
5.963623316594643
6.8664884500430015
7.906043210907698
9.102981779915218
10.481131341546853
12.067926406393289
13.894954943731374
15.998587196060573
18.420699693267164
21.209508879201906
24.420530945486497
28.11768697974231
32.374575428176435
37.27593720314938
42.91934260128778
49.417133613238335
56.89866029018293
65.51285568595509
75.43120063354615
86.85113737513521
100.0
```

```
#K-fold accuracy vs C
acc_vector = np.array(accuracy_iter2).mean(axis=1)
plt.plot(10**np.linspace(-1, 2), acc_vector)
plt.xscale("log")
```




```
hyperparam_vals[np.argmax(np.array(accuracy_iter2).mean(axis=1))]
```

```
1.6768329368110082
```

From the above plot, all values of `C` in [1, 100] seem to be optimal, and can be chosen as the optimal `C`!

Indeed, for any value of `C` in [1, 100], we get a similar test accuracy.

```
logreg = LogisticRegression(solver = 'newton-cg', C =1, max_iter=1000)
logreg.fit(X_train_poly, y_train)
X_test_poly = poly.fit_transform(X_test_scaled)
y_pred = logreg.predict(X_test_poly)

print(accuracy_score(y_pred, y_test)*100)
```

```
91.6955017301038
```

```python
logreg = LogisticRegression(solver = 'newton-cg', C =100, max_iter=1000)
logreg.fit(X_train_poly, y_train)
X_test_poly = poly.fit_transform(X_test_scaled)
y_pred = logreg.predict(X_test_poly)

print(accuracy_score(y_pred, y_test)*100)
```

```
91.62629757785467
```

D Datasets, assignment and project files

Datasets used in the book, assignment files, project files, and prediction problems report tempate can be found here

References