# DAMA61 - 1st assignment

## Exercise 1

### Code

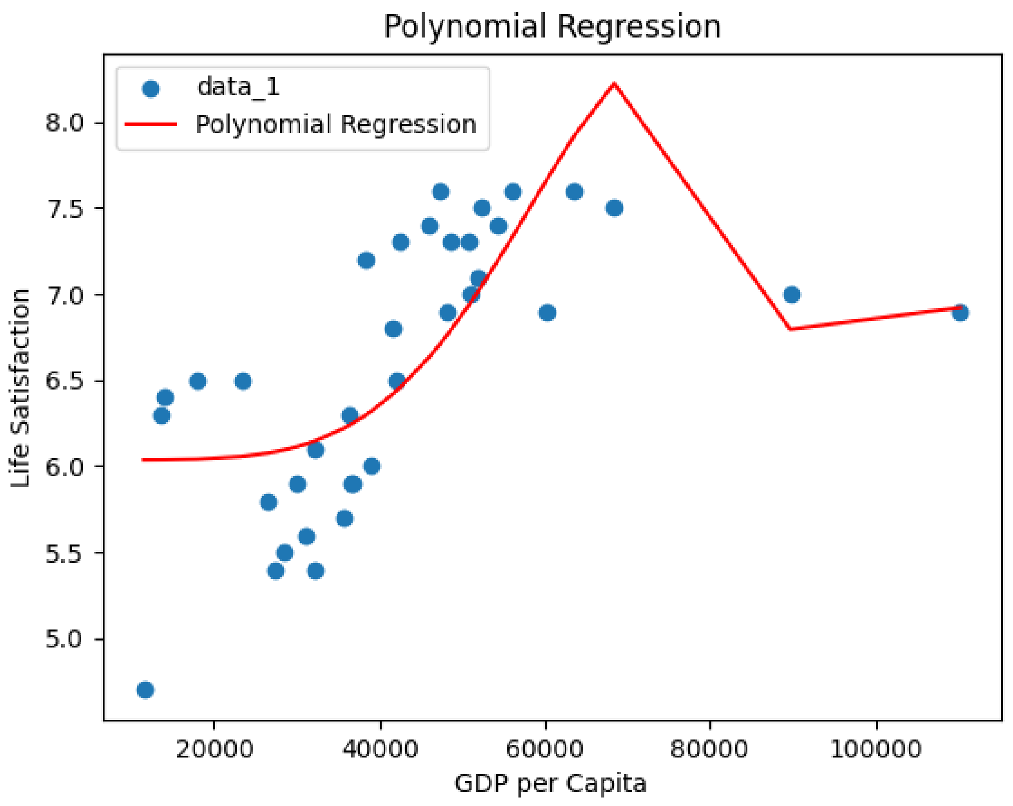
```
# Loading data
# Downloading and prepare the data
data_1 = "https://github.com/ageron/data/raw/main/"
lifesat = pd.read_csv(data_1 + "lifesat/lifesat_full.csv")

# Turning columns into arrays
X_1 = lifesat["GDP per capita (USD)"].values.reshape(-1, 1)
y_1 = lifesat["Life satisfaction"].values

# Making polynomial object with the degree 8
poly_1 = PolynomialFeatures(degree=8)

# Transforming polynomial to the independent variable X_1
X_1_poly = poly_1.fit_transform(X_1)

# Creating linear regression model
model_1 = LinearRegression()

# Fitting linear regression model to the polynomial features
model_1.fit(X_1_poly, y_1)

# Making the prediction of y_1 using the previously made polynomial
y_1_pred = model_1.predict(X_1_poly)

# Constructing the plot
plt.scatter(X_1, y_1, label='data_1')
plt.plot(X_1, y_1_pred, color='red', label='Polynomial Regression')
plt.legend()
plt.xlabel('GDP per Capita')
plt.ylabel('Life Satisfaction')
plt.title('Polynomial Regression')
plt.show()

GDP_to_predict = 97000

# Transforming GDP_to_predict into its polynomial representation
GDP_to_predict_poly = poly_1.transform([[GDP_to_predict]])

# Making the prediction of LSI for GDP using the polynomial generated in the
line above
LSI_predicted = model_1.predict(GDP_to_predict_poly)[0]
```

# DAMA61 - 1st assignment

```
print("Predicted LSI for GDP=97000 : ", LSI_predicted)
# Console output:
# Predicted LSI for GDP=97000 :  5.25018016958461

# Creating nearest neighbors model on X_1 with decided number of neighbors
three
nearest_neighbors = NearestNeighbors(n_neighbors=3).fit(X_1)
# Calculating distances and indices of the three nearest neighbors of the
given GDP value 97000
distances, indices = nearest_neighbors.kneighbors([[GDP_to_predict]])
# Extracting them to an array
nearest_neighbors_indices = indices[0]
# Using the array nearest_neighbors_indices to extract the corresponding LSI
values from y_1
LSI_neighbors = [y_1[i] for i in nearest_neighbors_indices]
# Estimating the mean
LSI_estimate = np.mean(LSI_neighbors)
print("Estimated LSI : ", LSI_estimate)
# Console output:
# Estimated LSI :  7.133333333333333

# We see that the two predictions we got from the previous questions are
different which was expected. Different modeling approaches bring different
results. The fact that the two numbers are very different is most likely
because we had to set the polynomial features degree to 8 as asked from the
exercise which ends up overfitting the model. Another approach would be a
different number of chosen neighbors in the kNN which also had to be 3 as
asked in the exercise.

# Using the exact same code as above in a for-loop for each degree to extract
the R-squared Value which measures the goodness of fit for each degree of the
polynomial regression model
degrees = range(1, 11)
r_squared_values = []

for degree in degrees:
   poly_1 = PolynomialFeatures(degree=degree)
   X_1_poly = poly_1.fit_transform(X_1)
   model_1 = LinearRegression()
   model_1.fit(X_1_poly, y_1)
   y_1_pred = model_1.predict(X_1_poly)
   r_squared = model_1.score(X_1_poly, y_1)
   r_squared_values.append(r_squared)

plt.plot(degrees, r_squared_values, marker='o')
plt.xlabel('Polynomial Degree')
plt.ylabel('R-squared Value')
plt.title('R-squared vs. Polynomial Degree')
plt.show()
# By appending it to a list and plot a graph it is visible that up to the 6th
degree of the polynomial the R-squared Value is at its highest, hence a good
fit for the model, but from there as the degrees rise the goodness of fit
lowers
```

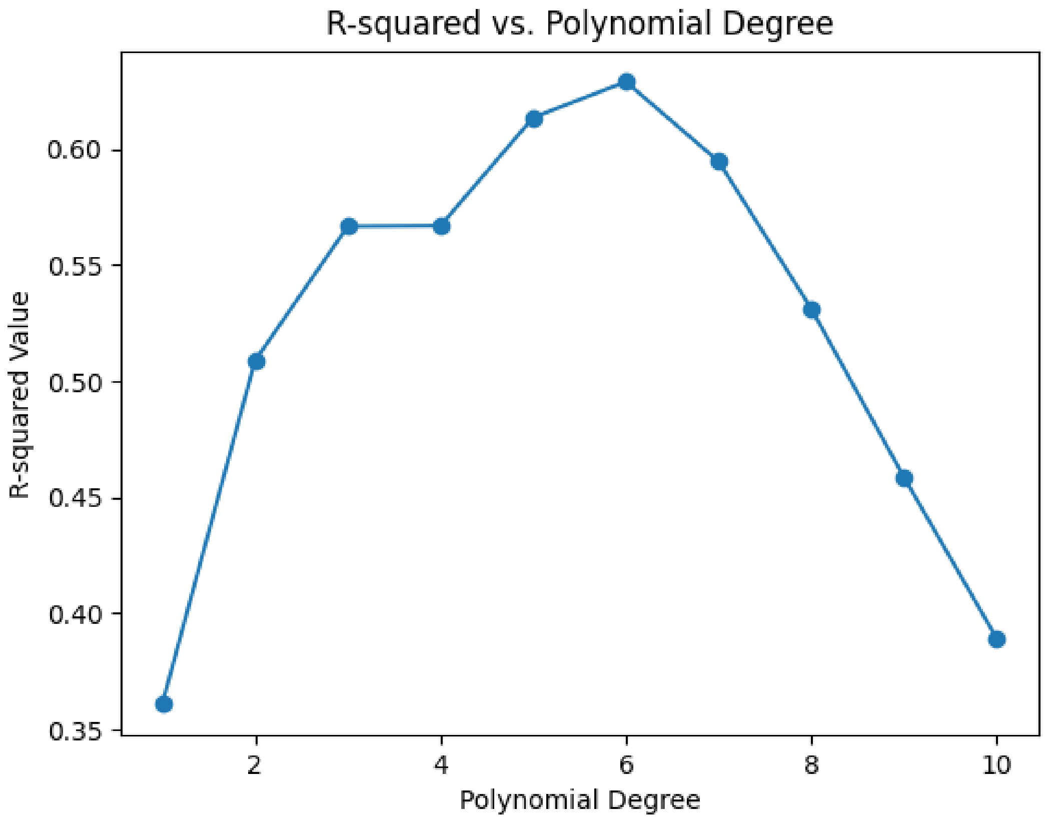
## Results

As we can see from the graph below, the Life Satisfaction has a trend to rise as the GDP rises up to 70k. After that it seems to have a trend to lower. While thinking about it, the first part seems reasonable, while the second part doesn't. This is an outcome that was due to outliers because as we can see in the graph, most of our data are around the uprising trend, while in the second part are very much less.



Predicted LSI for GDP=97000 :  5.25018016958461
Estimated LSI :  7.133333333333333

We see that the two predictions we got from the previous questions are different which was  expected. Different modeling approaches bring different results. The fact that the two numbers are very different is most likely because we had to set the polynomial features degree to 8 as asked from the exercise which ends up overfitting the model. Another approach would be a different number of chosen neighbors in the kNN which also had to be 3 as asked in the exercise.



The above graph shows the relationship between the polynomial degree and the R-squared value of our model. It is visible that up to the 6th degree of the polynomial the R-squared Value is at its highest, hence a good fit for the model, but from there as the degrees rise the goodness of fit lowers.

## Exercise 2

### Code

```
# Loading the data using the provided dataset
data_2 = fetch_california_housing(as_frame=True)

# Extracting features (X_2) and target (y_2)
X_2 = data_2.data
y_2 = data_2.target

# Defining the parameter grid for the Grid Search
param_grid = {'n_estimators': [5, 15, 25], 'max_features': [2, 4, 8]}

# Creating a Random Forest Regressor
rf_regressor = RandomForestRegressor()

# Creating the object to perform a Grid Search cross-validator
grid_search = GridSearchCV(estimator=rf_regressor, param_grid=param_grid,
cv=KFold(n_splits=3),
                          scoring='neg_mean_squared_error')

# Fitting the model to the data
grid_search.fit(X_2, y_2)

# Finding the best parameters
best_params = grid_search.best_params_
print("Best parameters : ", best_params)
# Console output:
# Best parameters :  {'max_features': 8, 'n_estimators': 25}

# Finding the best random forest regressor model
best_model = grid_search.best_estimator_
# Using the found best model to make predictions
y_2_pred = best_model.predict(X_2)
# Calculating the mean squared error between the actual target values
mse = mean_squared_error(y_2, y_2_pred)
# Calculating RMSE from the MSE
rmse = np.sqrt(mse)
print("RMSE : ", rmse)
# Console output:
# RMSE :  0.19878101905851503

# Getting feature importances from the best model
```

```
feature_importances = best_model.feature_importances_

# Creating a dataframe to associate feature names with their importances
feature_importance_df = pd.DataFrame({'Feature': X_2.columns, 'Importance':
feature_importances})

# Sorting the features by importance in descending order
sorted_feature_importance_df =
feature_importance_df.sort_values(by='Importance', ascending=False)

# Getting the top three most important features
top_features = sorted_feature_importance_df.head(3)
print("Top features : ")
print(top_features)
# Console output:
# Top features :
#      Feature  Importance
# 0     MedInc    0.522870
# 5   AveOccup    0.133947
# 6   Latitude    0.092459

# From the output of the above code we get: The parameters of the best model
being 'max_features': 8 and 'n_estimators': 25 by utilizing the functions of
GridSearchCV. An RMSE as low as 0.198 which means that the model is pretty
good at predicting housing prices in California. The top three most important
features of the model are MedInc, AveOccup and Latitude in that order from
most important to least important.
```

## Results

```
Best parameters :  {'max_features': 8, 'n_estimators': 25}
RMSE :  0.19878101905851503
Top features :
   Feature  Importance
0    MedInc    0.522870
5  AveOccup    0.133947
6  Latitude    0.092459
```

The parameters of the best model being 'max_features': 8 and 'n_estimators': 25 by utilizing the functions of GridSearchCV. An RMSE as low as 0.198 means that the model is pretty good at predicting housing prices in California. The top three most important features of the model are

MedInc, AveOccup and Latitude in that order from most important to least important.

## Exercise 3

### Code

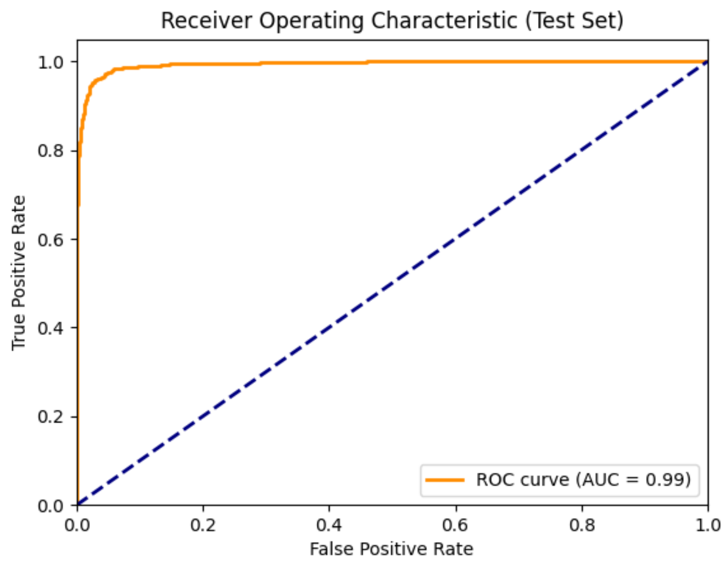
```
# Loading the MNIST dataset with 'auto' parser to avoid the irrelevant error
mnist = fetch_openml("mnist_784", parser='auto')
# Extracting the feature data on variable X_3 and target labels on variable
y_3)
X_3, y_3 = mnist.data, mnist.target.astype(int)

# Splitting data into training (6/7) and test (1/7) sets
# Setting a fixed random seed to keep the training and test sets the same at
multiple runs
X_3_train, X_3_test, y_3_train, y_3_test = train_test_split(X_3, y_3,
test_size=1 / 7, random_state=16)

# Creating binary labels: 1 for digit 4, 0 for other digits
y_3_train_binary = (y_3_train == 4)
y_3_test_binary = (y_3_test == 4)

# Creating a PCA object with 10 principal components for dimensionality
reduction.
# The optimal amount of principal components was found by multiple runs of the
project.
pca = PCA(n_components=10)
# Applying the pca on the training data
X_3_train_pca = pca.fit_transform(X_3_train)
# Transforming the test data
X_3_test_pca = pca.transform(X_3_test)

# Creating an SVM classifier
svm_cl = SVC()

# Training the SVM classifier on reduced training data
svm_cl.fit(X_3_train_pca, y_3_train_binary)

# Calculating predictions using cross-validation with 3 folds
cv_predictions = cross_val_predict(svm_cl, X_3_train_pca, y_3_train_binary,
cv=3)

# Calculating accuracy, precision, and recall using cross-validation results
```

```
accuracy = accuracy_score(y_3_train_binary, cv_predictions)
precision = precision_score(y_3_train_binary, cv_predictions)
recall = recall_score(y_3_train_binary, cv_predictions)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
# Console output:
#          Accuracy: 0.9815333333333334
#          Precision: 0.9322187897510449
#          Recall: 0.8746803069053708


# Calculating accuracy of the "not-4" guessing model
not_4_accuracy = 1 - np.mean(y_3_train_binary)
print("Accuracy of 'not-4' guessing model:", not_4_accuracy)
# Console output:
# Accuracy of 'not-4' guessing model: 0.90225

# Calculating the confusion matrix
confusion = confusion_matrix(y_3_train_binary, cv_predictions)

# Counting wrongly classified samples
wrong_4 = confusion[1][0]
wrong_not_4 = confusion[0][1]

print("Wrongly classified as 4s:", wrong_4)
print("Wrongly classified as non-4s:", wrong_not_4)
# Console output:
#          Wrongly classified as 4s: 735
#          Wrongly classified as non-4s: 373

# Predicting probabilities on the test set
y_3_prob_test = svm_cl.decision_function(X_3_test_pca)

# Calculating the ROC curve for the test set
fpr, tpr, _ = roc_curve(y_3_test_binary, y_3_prob_test)

# Calculating AUC for the test set
auc = roc_auc_score(y_3_test_binary, y_3_prob_test)

# Plotting the ROC curve for the test set
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (AUC =
{:.2f})'.format(auc))
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (Test Set)')
plt.legend(loc="lower right")
plt.show()

print("Test Set AUC:", auc)
# Console output:
#         Test Set AUC: 0.9935979287497957
```

## Results

Accuracy: 0.9815333333333334
Precision: 0.9322187897510449
Recall: 0.8746803069053708
Accuracy of 'not-4' guessing model: 0.90225
Wrongly classified as 4s: 735
Wrongly classified as non-4s: 373
Test Set AUC: 0.9935979287497957

The trained model achieved a high accuracy in correctly classifying digit 4, significantly outperforming a naive model that always guesses "not-4." However, there were still instances of misclassifications, with 735 samples wrongly classified as digit 4 and 373 samples wrongly classified as non-digit 4.


Receiver Operating Characteristic (Test Set)
True Positive Rate
False Positive Rate
ROC curve (AUC = 0.99)