COS30049 – Computing Technology Innovation Project

Assignment 2 – AI-based Project

Leo Holden – 103996982

Jackson Webb – 103572997

Declan Hargreaves - 104300584

Submission Date: 29/09/2024

Project Environment Instructions

Creating and Exporting Environment

- First in conda create an environment.

  conda create -n assignment2env

- Activate the newly created environment and install all dependencies. The packages that need to be installed are the following: pandas, scikit-learn, xgboost, numpy, matplotlib and seaborn.

  conda activate assignment2env

  conda install [package name]

- Export the current environment.

  conda env export > assignment2env.yml

Importing Environment

- Open the conda CLI and refer to the assignment2env.yml file. Place this file inside the location specified by the user prompt i.e C:/Users/YourName. Once the file is inside this location run the following command.

  conda env create -f assingment2env.yml

Training Instructions

Weather Dataset Preprocessing

- First load the csv. Pandas provides a function to read a specified csv allowing us to read and manipulate data.

  weather_data = pd.read_csv('./weatherAUS copy.csv)

- With the csv data now loaded, we can drop irrelevant data
- The weather dataset contains 23 columns, a majority of which are to be dropped from our dataset. Using the ‘drop’ function we can specify which columns to remove, an example of which is provided below.

  weather_data.drop([‘Rainfall’], axis=1, inplace=True)

- This is repeated for the following columns: Evaporation, Sunshine, WindGustDir, WindGustSpped, WinDir9am, WinDir3pm, WindSpeed9am, WindSpeed3pm, Humidity9am, Humidity3pm, Pressure9am, Pressure3pm, Cloud9am, Cloud3pm, Temp9am, Temp3pm, RainToday, RainTomorrow.
- Leaving 4 remaining columns, Date, MinTemp, MaxTemp and location. The MinTemp and MaxTemp columns contain a small amount of missing date which is required to be filled in.
- The function below transforms a locations MinTemp and MaxTemp if the data is missing, replacing it with the median for that location.

  weather_data['MinTemp'] =
  weather_data.groupby('Location')['MinTemp'].transform(lambda x: x.fillna(x.median()))

  weather_data['MaxTemp'] =
  weather_data.groupby('Location')['MaxTemp'].transform(lambda x: x.fillna(x.median()))

- Next, we need to insert columns we want to train on into a data frame. This requires a dictionary of data columns and a name to reference the column by.

  ```
  data = {
     'Date': weather_data['Date'],
     'Location': weather_data['Location'],
     'MinTemp': weather_data['MinTemp'],
     'MaxTemp': weather_data['MaxTemp']
  }
  weather_df = pd.DataFrame(data)
  ```

- With the data now contained in a data frame, we want this data frame to hold only Melbourne locations. This is achieved by iterating through the ‘Location’ column and only storing the data if the location is ‘Melbourne’ or ‘MelbourneAirport’.

  melb_weather_df = weather_df[(weather_df['Location'] == 'Melbourne') |
  (weather_df['Location'] == 'MelbourneAirport')]

- Our dataframe now holds 4 columns, ‘Date’, ‘Location’, ‘MinTemp’, ‘MaxTemp’. With these columns only containing data from Melbourne.
- It is also useful to convert our date data to datetime and transform the location names to integers.

```
melb_weather_df['Date'] = pd.to_datetime(melb_weather_df['Date'], format='%Y-%m-%d')
```

```
label_encoder = LabelEncoder()

melb_weather_df.loc[:, 'Location'] =
label_encoder.fit_transform(melb_weather_df['Location'])
```

Weather Dataset Training and Prediction

- To train our model we need to specify the target and feature variables as well as split that data. As we are training this current model on the ‘MinTemp’ we set the y (Target) variable to the ‘MinTemp’ column. For the features we want to drop all other columns except ‘Location’. The X (Features) variable contains the ‘Location’ data.

```
X = melb_weather_df.drop(['MinTemp', ‘MaxTemp’, ‘Date’], axis=1)

y = melb_weather_df['MinTemp']
```

- Split the data. The test_size parameter determines the percentage of the data set that will be split into training/validation data. In the first split 20% of the data is training data and the second split 25% of the data is validation data.

```
X_train, X_test, y_train_max, y_test_max = train_test_split(X, y_max, test_size=0.2,
random_state=42)

X_train, X_val, y_train_max, y_val_max = train_test_split(X_train, y_train_max,
test_size=0.25, random_state=42)
```

- We have used multiple models to train and predict for this data set. The first being random forest regression, as shown below. The n_estimators parameter controls how many decision trees the forest uses.

rf = RandomForestRegressor(n_estimators=100, random_state=42)

rf.fit(X_train, y_train)

- The second model is a gradient boosting regression algorithm, also shown below. n_estimators in this context relates to the amount of boosting iterations occur.

gb_max = GradientBoostingRegressor(n_estimators=100, random_state=42)

gb_max.fit(X_train, y_train_max)

- With the models now trained, we can predict using the X validation data.

y_predict_gb = gb.predict(X_val)

y_predict_rf_max = rf_max.predict(X_val)

- Finally, we can calculate the model's performance using the mean squared error as well as mean absolute error. The squared=False parameter differentiates between root mean squared error and mean squared error, with false being RMSE and true being MSE.

rmse_gb = mean_squared_error(y_val, y_predict_gb, squared=False)

mse_gb = mean_squared_error(y_val, y_predict_gb, squared=True)

mae_gb = mean_absolute_error(y_val, y_predict_gb)

print(f"RMSE - GB MinTemp: {rmse_gb}")

print(f"MSE - GB MinTemp: {rmse_gb}")

print(f"MAE - GB MinTemp: {mae_gb}")

Influenza Dataset Preprocessing

- Load the dataset.

```
file_path = './influenza_weekly.csv'

influenza_data = pd.read_csv(file_path)
```

- For this data set we only want data pertaining to the Oceanic, Melanesian and Polynesian flu region. This is achieved by dropping any data that doesn't conform to the specific regions.

```
influenza_data = influenza_data[influenza_data['FLUREGION'] == 'Oceania Melanesia Polynesia']
```

- Next, we want to extract the year, month and day from each date while also formatting the date column.

```
influenza_data['date'] = pd.to_datetime(influenza_data['Year'].astype(str) + influenza_data['Week'].astype(str).str.zfill(2) + '1', format='%Y%W%w')

influenza_data.loc[:, 'year'] = influenza_data['date'].dt.year

influenza_data.loc[:, 'month'] = influenza_data['date'].dt.month

influenza_data.loc[:, 'dayofyear'] = influenza_data['date'].dt.dayofyear
```

- The column for the total amount of cases reported for that week is inconveniently named, and as such we must rename it.

```
influenza_data['cases'] = influenza_data['ALL_INF']
```

- The data set contains information relating to influenza outbreak type. This information will be useful when plotting data and identifying trends, so we want to transform it into a more workable data type. To do this we must transform the outbreak type with an integer.

```
influenza_data['title'] = influenza_data['TITLE'].replace({'No Activity': 0, 'Sporadic': 1,
                'Local Outbreak' : 2, 'Regional Outbreak' : 3, 'Widespread Outbreak' : 4})
```

- Likewise, we want to add seasons to our data, allowing us to identify seasonal trends and train with this data. To do this, for each row of data we map a month to a season and add

this data to the season's column. 0 being Summer, 1 being Autumn, 2 being Winter and 3 being Spring.

```
def get_season(month):
    if month in [12, 1, 2]:
        return 0
    elif month in [3, 4, 5]:
        return 1
    elif month in [6, 7, 8]:
        return 2
    else:
        return 3

influenza_data['season'] = influenza_data['month'].apply(lambda x : get_season(x))
```

Influenza Dataset Training and Prediction

- Again, we must set the features and target variables.

```
features = [ 'month', 'year', 'season', 'title']
GBX = influenza_data[features]
GBy = influenza_data['cases']
```

- Next, split the data.

```
GBX_train, GBX_test, GBy_train, GBy_test = train_test_split(GBX, GBy, test_size=0.2, random_state=42)
```

- For this data set 4 models have been trained. A gradient boosting regressor, a random forest regressor, an XGBoost classifier and an XGBoost regressor. The random forest, XGBoost classifer and gradient booster follow the same training and prediction code with different evaluation methods.
- First train the model's.

```
model = GradientBoostingRegressor(n_estimators=1000, random_state=42)
model.fit(GBX_train, GBy_train)
```

- Predict using the test data.

```
GBy_pred = model.predict(GBX_test)
```

- Finally, we can view the model's performances. Particularly the RMSE and R^2 score for the random forest regressor and gradient booster model's. For the random forest classifier, we use the accuracy, precision, recall and f1 scores. All models display feature importance.

```
GBrmse = mean_squared_error(GBy_test, GBy_pred, squared=False)
GBr2 = r2_score(GBy_test, GBy_pred)
print(f"Random Forest root Mean squared error: {GBrmse}")
print(f"Random Forest R-squared score: {GBr2}")

for feature, importance in zip(features, model.feature_importances_):
    print(f"{feature}: {importance}")

accuracy = accuracy_score(RFy_test, y_pred)
precision = precision_score(RFy_test, y_pred, average='weighted')
recall = recall_score(RFy_test, y_pred, average='weighted')
f1 = f1_score(RFy_test, y_pred, average='weighted')

print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
```

- For the XGBoost regressor is similar with some slight differences. The feature and target selection is the same, as well using the same split code.
- To create the XGB regressor, do the following. The objective parameter specifies the learning objective with reg:squarederror pertaining to regression with squared loss.

model = XGBRegressor(objective='reg:squarederror', n_estimators=1000, max_depth=3, random_state=42)

- Then repeat the fit, predict and model evaluation steps as previously shown using the XGBoost model.

Electricity Dataset Preprocessing

- Load the electricity dataset

electricity_data = pd.read_csv('electricity_data.csv')

- This data set contained date, minimum temperature and maximum temperature columns that needed to have extra data extracted from them. First convert the date to datetime then extract the year, month and day from the date.

electricity_data['date'] = pd.to_datetime(electricity_data['date'], format='%Y-%m-%d')

electricity_data.loc[:, 'year'] = electricity_data['date'].dt.year

electricity_data.loc[:, 'month'] = electricity_data['date'].dt.month

electricity_data.loc[:, 'dayofyear'] = electricity_data['date'].dt.dayofyear

- Next, extract an average temperature and add that to the ‘avg_temp’ column.

electricity_data['avg_temp'] = (electricity_data['min_temperature'] + electricity_data['max_temperature']) / 2

- The ‘holday’ and ‘school_day’ columns can provide us with helpful data when identifying trends withing power consumption but first they need to be transformed into integers. The format of these columns is Y/N and we want to replace those with 0 for N and 1 for Y.

electricity_data['holiday'] = electricity_data['holiday'].replace({'Y': 1, 'N': 0})

electricity_data['school_day'] = electricity_data['school_day'].replace({'Y': 1, 'N': 0})

- Similarly, the solar exposure could provide insight into power usage trends but contains missing data. We want to remove this missing data.

electricity_data.dropna(subset=['solar_exposure'], inplace=True)

Electricity Dataset Training and Prediction

- First, set the feature and target variables. The demand data needs to be of type float so we change all demand values to floats.

features = ['avg_temp', 'holiday', 'solar_exposure', 'dayofyear','year','month']

RFX = electricity_data[features]

RFy = electricity_data['demand'].values.astype("float64")

- Split the data.

RFX_train, RFX_test, RFy_train, RFy_test = train_test_split(RFX, RFy, test_size=0.2, random_state=42)

- This data set was trained on 3 models, a random forest regressor, an XGBoost regressor and an XGBoost classifier. The random forest regressor and XGBoost regressor utilise the same code with one minor difference in the creation of the models. The XGB regressor requires us to specify the learning objective. Regression with squared loss was chosen.

model = XGBRegressor(objective='reg:squarederror', n_estimators=10, max_depth=2, random_state=42)

model.fit(GBRX_train, GBRy_train)

model = RandomForestRegressor(n_estimators=100, random_state=42)

model.fit(RFX_train, RFy_train)

- Train the model using each model's on the training data.

model.fit(RFX_train, RFy_train)

- Predict using each model's test data.

y_pred = model.predict(RFX_test)

- Finally, evaluate the model's performance using the RMSE and R^2 score as well as print out the feature importance.

RFrmse = mean_squared_error(RFy_test, y_pred, squared=False)

RFr2 = r2_score(RFy_test, y_pred)

print(f"Random Forest root Mean squared error: {RFrmse}")

print(f"Random Forest R-squared score: {RFr2}")

for feature, importance in zip(features, model.feature_importances_):

print(f"{feature}: {importance}")

- The XGBoost classifier requires similar steps. First set the features/target and split the data as before. We then need to label encode the trained demand data.

encoder = LabelEncoder()

GBy_train = encoder.fit_transform(GBy_train)

- Next, create the XGB classifier

boost = XGBClassifier(max_depth=3)

- Train the model on the training data and predict using the test data as shown previously. Then evaluate the model's performance.