Final Project: Building a Rainfall Prediction Classifier

Estimated time needed: **30** minutes

Objectives

After completing this lab you will be able to:

- Explore and perform feature engineering on a real-world data set
- Build a classifier pipeline and optimize it using grid search cross validation
- Evaluate your model by interpreting various preofmrance metrics and visualizations

Instructions

TBW

About The Dataset

The original source of the data is Australian Government's Bureau of Meteorology and the latest data can be gathered from http://www.bom.gov.au/climate/dwo/.

The dataset you'll use in this project was downloaded from Kaggle at [https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package?resource=download&select=weatherAUS.csv]
Column definitions were gathered from http://www.bom.gov.au/climate/dwo/IDCJDW0000.shtml

The dataset contains observations of weather metrics for each day from 2008 to 2017, and includes the following fields:

Field	Description	Unit	Type
Date	Date of the Observation in YYYY-MM-DD	Date	object
Location	Location of the Observation	Location	object
MinTemp	Minimum temperature	Celsius	float
MaxTemp	Maximum temperature	Celsius	float
Rainfall	Amount of rainfall	Millimeters	float

Field	Description	Unit	Type
Evaporation	Amount of evaporation	Millimeters	float
Sunshine	Amount of bright sunshine	hours	float
WindGustDir	Direction of the strongest gust	Compass Points	object
WindGustSpeed	Speed of the strongest gust	Kilometers/Hour	object
WindDir9am	Wind direction averaged over 10 minutes prior to 9am	Compass Points	object
WindDir3pm	Wind direction averaged over 10 minutes prior to 3pm	Compass Points	object
WindSpeed9am	Wind speed averaged over 10 minutes prior to 9am	Kilometers/Hour	float
WindSpeed3pm	Wind speed averaged over 10 minutes prior to 3pm	Kilometers/Hour	float
Humidity9am	Humidity at 9am	Percent	float
Humidity3pm	Humidity at 3pm	Percent	float
Pressure9am	Atmospheric pressure reduced to mean sea level at 9am	Hectopascal	float
Pressure3pm	Atmospheric pressure reduced to mean sea level at 3pm	Hectopascal	float
Cloud9am	Fraction of the sky obscured by cloud at 9am	Eights	float
Cloud3pm	Fraction of the sky obscured by cloud at 3pm	Eights	float
Temp9am	Temperature at 9am	Celsius	float
Temp3pm	Temperature at 3pm	Celsius	float
RainToday	If there was at least 1mm of rain today	Yes/No	object
RainTomorrow	If there is at least 1mm of rain tomorrow	Yes/No	object

Install and import the required libraries

```
# !pip install numpy
# !pip install pandas
# !pip install matplotlib
# !pip install scikit-learn

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split, GridSearchCV,
cross_val_score, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.metrics import classification_report, confusion_matrix,
ConfusionMatrixDisplay
```

Load the data

```
df = pd.read_csv("weatherAUS_2.csv")
df.head()
```

Drop all rows with missing values

To try to keep things simple we'll drop missing values and see what's left

```
df = df.dropna()
df.info()
```

Since we still have 56k observations left after dropping missing values, we may not need to impute any missing values.
Let's see how we do.

```
df.columns
```

Exercise 1. Data leakage considerations

Are there any barriers to being able to predict whether it will rain tomorrow given the available data?

Exercise 1 Response

If your goal is to predict attributes for tomorrow's weather, using data that relies on complete information from today's conditions is impractical. For instance, actual minimum and maximum temperatures for today cannot be determined until the day is complete, typically at midnight. Similarly, features such as RainToday, Rainfall, Evaporation, Sunshine, WindGustDir, and WindGustSpeed are not usable because they reflect the full day's conditions.

Attributes recorded at specific times, like 9 am and 3 pm, are also restricted; using both would mean your prediction needs to be made after 3 pm.

These factors highlight potential data leakage issues. Using data in training that would not be accessible in a real-world scenario for predicting tomorrow's rainfall can lead to misleading results.

However, if we adjust our approach and aim to predict today's rainfall using historical weather data up to and including yesterday, we can utilize all available features. This shift could be particularly useful for practical applications, such as deciding whether it's wise to bike to work.

With this new target, we should update the names of the rain columns accordingly.

```
df = df.rename(columns={'RainToday': 'RainYesterday',
                        'RainTomorrow': 'RainToday'
                        })
```

Exercise 2. Data granularity

Consider the Location field.
Do you think this all of the locations are useful information to include?
Any ideas on what to do to proceed?

Exercise 2 Response

Would the weather patterns have the same predictability in vastly different locations in Australia? I would think not.
The chance of rain in one location can be much higher than in another. Using all of the locations requires a more complex model as it needs to adapt to local weather patterns.
Let's see how many observations we have for each location.

Location selection

You could do some research to group cities in the `Location` column by distance, which I've done for you behind the scenes.
I found that Watsonia is only 15 km from Melbourne, and the Melbourne Airport is only 18 km from Melbourne.
Let's group these three locations together and use only their weather data to build our localized prediction model.
Because theere might still be some slight variations in the weather patterns we'll keep `Location` as a categorical variable.

```
df =
df[df.Location.isin(['Melbourne','MelbourneAirport','Watsonia',])]
df. info()
```

We still have 7557 records, which should be enough to build a reasonably good model.
You could always gather more data if needed by updating it from the source.

Extracting a seasonality feature

Now consider the `Date` column. We would expect the weather patterns to be seasonal, having different predictablitiy levels in winter and summer for example.
There may be some variation with `Year` as well, but we'll leave that for now. Let's engineer a `Season` feature from `Date` and drop `Date` afterward.
An easy way to do this is to define a function that assigns seasons to given months, then use that function to transform the `Date` column.

```
# Create a function to map dates to seasons
def date_to_season(date):
    month = date.month
    day = date.day

    if (month == 12) or (month == 1) or (month == 2):
        return 'Summer'
    elif (month == 3) or (month == 4) or (month == 5):
        return 'Autumn'
    elif (month == 6) or (month == 7) or (month == 8):
        return 'Winter'
    elif (month == 9) or (month == 10) or (month == 11):
        return 'Spring'

# Convert the 'Date' column to datetime format
df['Date'] = pd.to_datetime(df['Date'])

# Apply the function to the 'Date' column
df['Season'] = df['Date'].apply(date_to_season)

df=df.drop(columns=['Date'])
df
```

Looks like we have a good set of features to work with.

Let's go ahead and build our model.

But wait, let's take a look at how well balanced our target is.

```
df['RainToday'].value_counts()
```

Exercise 3. What can you conclude from these counts?

- How often does it rain annualy in the Melbourne area?
- How accurate would you be if you just assumed it won't rain every day?
- Is this a balanced dataset?
- Next steps?

Exercise 3 Response

Apparently it is usally not raining in the Melbourne area, although it rains on average about 30% of the days in a year.
Consider that if your model was super simplistic and always predicted that it will not rain today. You would be correct about 70% of the time. That's not a bad guess!
This is an important reminder that baseline accuracy isn't necessarily a good performance indicator for imbalanced data.
A model that always predicts the majority class might show high accuracy but lack real predictive power.

Stratification is particularly critical when working with unbalanced data, as it helps the model learn from both classes adequately and ensures that the evaluation is fair, preventing overly optimistic results due to an imbalanced distribution in the splits. Since we have unbalanced classes in our target, we need to stratify the target so it has the same proportion of classes in in both the training and the testing sets.

```
### Exercise 4. Define the feature and target dataframes
# Complete the followng code:
# X = df.drop(columns='', axis=1)
# y = df['']

### Exercise 4 Response
X = df.drop(columns='RainToday', axis=1)
y = df['RainToday']


### Exercise 5. Split data into training and test sets, ensuring
target stratification
# Complete the followng code:
#  X_train, X_test, y_train, y_test = train_test_split(..., ...,
test_size=0.2, stratify=..., random_state=42)

### Exercise 5 Response
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, stratify=y, random_state=42)


# Automatically detect numerical and categorical columns and assign
them to separate numeric and categorical features
numeric_features =
X_train.select_dtypes(include=['number']).columns.tolist()
categorical_features = X_train.select_dtypes(include=['object',
'category']).columns.tolist()


## Define preprocessing pipelines for both feature types

### Scale the numeric features
# Preprocessing pipeline for numeric features
```

```
numeric_transformer = Pipeline(steps=[('scaler', StandardScaler())])

### One-hot encode the categoricals
# Preprocessing pipeline for categorical features
categorical_transformer = Pipeline(steps=[('onehot',
OneHotEncoder(handle_unknown='ignore'))])


### Combine the transformers into a single column transformer
# We'll use the sklearn "column transformer" estimator to separately
transform the features, which will then concatenate the output as a
single feature space.
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

### Now let's create a pipeline by combining the preprocessing with a
Random Forest classifier
pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Define a parameter grid to use in a cross validation grid search
model optimizer
param_grid = {
    'classifier__n_estimators': [50, 100],
    'classifier__max_depth': [None, 10, 20],
    'classifier__min_samples_split': [2, 5]
}

### Pipeline usage in crossvalidation
# Recall that the pipeline is repeatedly used within the
crossvalidation to fit on each internal training fold and predict on
its corresponding validation fold

## Perform grid search cross-validation and fit the best model to the
training data
### Select a cross-validation method, ensuring target stratification
during validation

scv = StratifiedKFold(n_splits=5, shuffle=True)

## Exercise 6. Instantiate and fit GridSearchCV to the pipeline

# Complete the followng code:
# grid_search = GridSearchCV(..., param_grid, cv=...,
scoring='accuracy', verbose=2)
# grid_search.fit(..., ...)

### Exercise 6 Response
grid_search = GridSearchCV(pipeline, param_grid, cv=scv,
scoring='accuracy', verbose=2)
grid_search.fit(X_train, y_train)


### Print the best parameters and best crossvalidation score
print("\nBest parameters found: ", grid_search.best_params_)
print("Best cross-validation score:
{:.2f}".format(grid_search.best_score_))

## Exercise 7. Display your model's estimated score
# Complete the followng code:
# test_score = grid_search.score(X_test, y_test)
# print("Test set score: {:.2f}".format(test_score))

### Exercise 7 Response
test_score = grid_search.score(X_test, y_test)
print("Test set score: {:.2f}".format(test_score))
```

So we have a reasonably accurate classifer, which is expected to correctly predict about 84% of the time whether it will rain today.
Let's take a deeper look at the results.

The best model is stored within the gridsearch object.

```
### Exercise 8. Get the model predictions from the grid search
estimator on the unseen data
# Complete the followng code:
# y_pred = grid_search.predict(...)

### Exercise 8 Response
y_pred = grid_search.predict(X_test)

# Exercise 9. Print the classification report
# Complete the followng code:
# print("\nClassification Report:")
# print(...(y_test, y_pred))

### Exercise 9 Response
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

### Exercise 10. Plot the confusion matrix
# Complete the followng code:
# conf_matrix = ...(y_test, y_pred)
# disp = ConfusionMatrixDisplay(confusion_matrix=...)
# disp.plot(cmap='Blues')
# plt.title('Confusion Matrix')
# plt.show()

### Exercise 10 Response
conf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix)
disp.plot(cmap='Blues')
plt.title('Confusion Matrix')
plt.show()
```

Feature importances

Recall that to obtain the categorical feature importances, we have to work our way backward through the modelling pipeline to associate the feature importances with their original input variables, not the one-hot encoded ones. We don't need to do this for the numeric variables because we didn't modify their names in any way.
Remmeber we went from categorical features to one-hot encoded features, using the 'cat' column transformer

```
grid_search.best_estimator_['preprocessor'].named_transformers_['cat']
.named_steps['onehot']

# Cool! Now let's get all of the feature importances and associate
them with the original features

### Exercise 11. Extract the feature importances
# Complete the followng code:
# feature_importances = grid_search.best_estimator_['classifier']. ...

### Exercise 11 Response
feature_importances =
grid_search.best_estimator_['classifier'].feature_importances_


# Combine numeric and categorical feature names
feature_names = numeric_features +
list(grid_search.best_estimator_['preprocessor']
                                        .named_transformers_['cat']
                                        .named_steps['onehot']
                                        .get_feature_names_out(categor
ical_features))

# Define a ranked feature importance DataFrame
importance_df = pd.DataFrame({'Feature': feature_names,
                              'Importance': feature_importances
                             }).sort_values(by='Importance',
ascending=False)

N = 20  # Change this number to display more or fewer features
```

```
top_features = importance_df.head(N)

# Plotting
plt.figure(figsize=(10, 6))
plt.barh(top_features['Feature'], top_features['Importance'],
color='skyblue')
plt.gca().invert_yaxis()  # Invert y-axis to show the most important
feature on top
plt.title(f'Top {N} Most Important Features in predicting whether it
will rain today')
plt.xlabel('Importance Score')
plt.show()
# Print test score summary
print(f"\nTest set accuracy: {test_score:.2%}")

# Now we can get the feature names from the one-hot encoder
grid_search.best_estimator_['preprocessor'].named_transformers_['cat']
.named_steps['onehot'].get_feature_names_out(categorical_features)
```

from https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package?resource=download&select=weatherAUS.csv

image.png