# Data Uploading

```
from IPython import get_ipython
from IPython.display import display
# Initialize Spark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MySparkApp").getOrCreate()

# File location and type
file_location = "weatherHistory.csv"
file_type = "csv"

# CSV options
infer_schema = "false"
first_row_is_header = "false"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)
```

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Initialize Spark session
spark = SparkSession.builder.master("local[*]").appName("WeatherAnalysis").getOrCreate()

# Load data into DataFrame
file_path = "weatherHistory.csv"
weather_df = spark.read.csv(file_path, header=True, inferSchema=True)

# Show schema to understand data types
weather_df.printSchema()

# Show a sample of the data
weather_df.show(5)
```

```
---+------------------------+--------+------------------+----------------------+------------------+----------+--------------------+-
(C)|Apparent Temperature (C)|Humidity| Wind Speed (km/h)|Wind Bearing (degrees)|   Visibility (km)|Loud Cover|Pressure (millibars)|
---+------------------------+--------+------------------+----------------------+------------------+----------+--------------------+-
221|      7.3888888888888875|    0.89|           14.1197|                 251.0|15.826300000000002|       0.0|             1015.13|P
558|       7.227777777777776|    0.86|           14.2646|                 259.0|15.826300000000002|       0.0|             1015.63|P
778|       9.377777777777778|    0.89|3.9284000000000003|                 204.0|           14.9569|       0.0|             1015.94|P
889|       5.944444444444446|    0.83|           14.1036|                 269.0|15.826300000000002|       0.0|             1016.41|P
553|       6.977777777777779|    0.83|           11.0446|                 259.0|15.826300000000002|       0.0|             1016.51|P
---+------------------------+--------+------------------+----------------------+------------------+----------+--------------------+-
```

# Data Cleaning

Null Value Removal

```
# Convert formatted date to DateType
weather_df = weather_df.withColumn("Formatted Date", col("Formatted Date").cast("timestamp"))
```

```
# Handle any missing values (drop rows with null values for simplicity)
weather_df = weather_df.dropna()

# Verify cleaned data
weather_df.show(5)
```

```
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
|     Formatted Date|      Summary|Precip Type|  Temperature (C)|Apparent Temperature (C)|Humidity| Wind Speed (km/h)|Wind Bearing
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
|2006-03-31 22:00:00|Partly Cloudy|       rain|9.472222222222221|      7.3888888888888875|    0.89|           14.1197|
|2006-03-31 23:00:00|Partly Cloudy|       rain|9.355555555555558|       7.227777777777776|    0.86|           14.2646|
|2006-04-01 00:00:00|Mostly Cloudy|       rain|9.377777777777778|       9.377777777777778|    0.89|3.9284000000000003|
|2006-04-01 01:00:00|Partly Cloudy|       rain| 8.28888888888889|       5.944444444444446|    0.83|           14.1036|
|2006-04-01 02:00:00|Mostly Cloudy|       rain|8.755555555555553|       6.977777777777779|    0.83|           11.0446|
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
only showing top 5 rows
```

Date Formatting

```
from pyspark.sql.functions import year, month, dayofmonth, hour

# Extract time-based features
weather_df = weather_df.withColumn("Year", year("Formatted Date")) \
                       .withColumn("Month", month("Formatted Date")) \
                       .withColumn("Day", dayofmonth("Formatted Date")) \
                       .withColumn("Hour", hour("Formatted Date"))

# Show the transformed data
weather_df.show(5)
```

```
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
|     Formatted Date|      Summary|Precip Type|  Temperature (C)|Apparent Temperature (C)|Humidity| Wind Speed (km/h)|Wind Bearing
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
|2006-03-31 22:00:00|Partly Cloudy|       rain|9.472222222222221|      7.3888888888888875|    0.89|           14.1197|
|2006-03-31 23:00:00|Partly Cloudy|       rain|9.355555555555558|       7.227777777777776|    0.86|           14.2646|
|2006-04-01 00:00:00|Mostly Cloudy|       rain|9.377777777777778|       9.377777777777778|    0.89|3.9284000000000003|
|2006-04-01 01:00:00|Partly Cloudy|       rain| 8.28888888888889|       5.944444444444446|    0.83|           14.1036|
|2006-04-01 02:00:00|Mostly Cloudy|       rain|8.755555555555553|       6.977777777777779|    0.83|           11.0446|
+-------------------+-------------+-----------+-----------------+------------------------+--------+------------------+------------
only showing top 5 rows
```

```
# Example of creating lagged features
from pyspark.sql import Window
from pyspark.sql.functions import lag

# Define a window partitioned by year
windowSpec = Window.partitionBy("Year").orderBy("Formatted Date")

# Create lagged feature (previous day's temperature)
weather_df = weather_df.withColumn("Prev_Temperature", lag("Temperature (C)").over(windowSpec))

# Show the DataFrame with lagged feature
weather_df.show(5)
```

```
+-------------------+-------------+-----------+------------------+------------------------+--------+------------------+------------
|     Formatted Date|      Summary|Precip Type|   Temperature (C)|Apparent Temperature (C)|Humidity| Wind Speed (km/h)|Wind Bearing
+-------------------+-------------+-----------+------------------+------------------------+--------+------------------+------------
|2006-01-01 00:00:00|Mostly Cloudy|       rain| 1.161111111111113|      -3.238888888888888|    0.85|           16.6152|
|2006-01-01 01:00:00|Mostly Cloudy|       rain|1.6666666666666667|     -3.1555555555555554|    0.82|20.253800000000002|
|2006-01-01 02:00:00|     Overcast|       rain|1.7111111111111101|      -2.194444444444444|    0.82|             14.49|
|2006-01-01 03:00:00|Mostly Cloudy|       rain|1.1833333333333347|     -2.7444444444444454|    0.86|           13.9426|
|2006-01-01 04:00:00|Mostly Cloudy|       rain|1.2055555555555566|      -3.072222222222223|    0.85|15.906800000000002|
+-------------------+-------------+-----------+------------------+------------------------+--------+------------------+------------
only showing top 5 rows
```

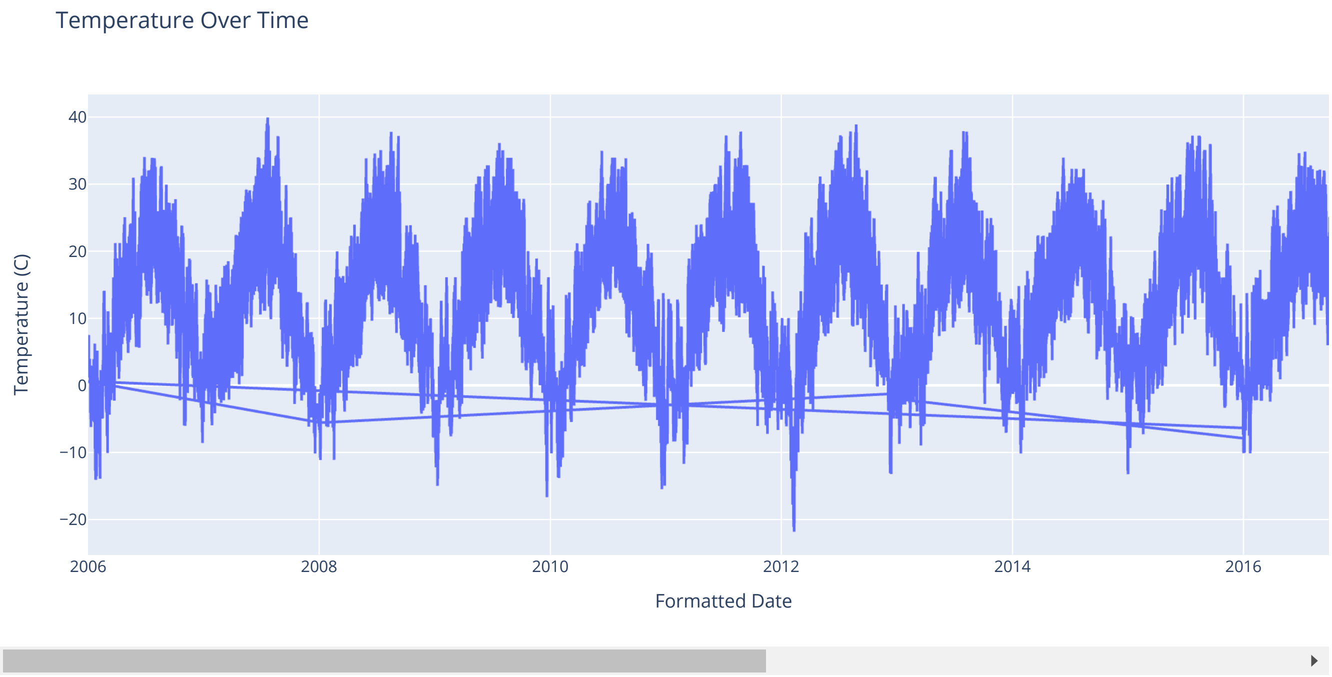
```
# Example using ARIMA in a Python environment (use `prophet` or `statsmodels` in Colab/Databricks)
from statsmodels.tsa.arima.model import ARIMA

# Extract necessary columns for time series forecasting (you would need to convert the data to Pandas format)
weather_df_pandas = weather_df.toPandas()

# Build ARIMA model on temperature data (example, after preprocessing)
```

```
model = ARIMA(weather_df_pandas['Temperature (C)'], order=(5,1,0))  # Example ARIMA(5,1,0)
model_fit = model.fit()

# Make predictions
forecast = model_fit.forecast(steps=12)  # Forecasting next 12 months
print(forecast)
```

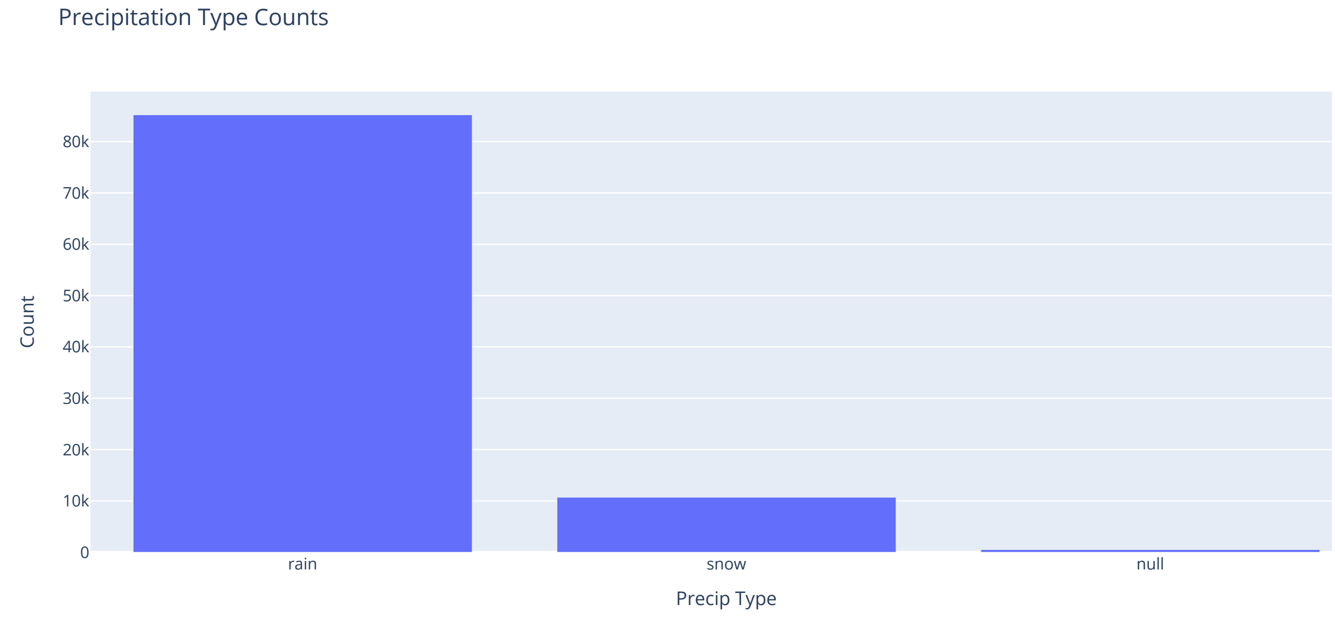
```
96453   -0.759531
96454   -0.806908
96455   -0.823332
96456   -0.819081
96457   -0.805952
96458   -0.794251
96459   -0.782955
96460   -0.774688
96461   -0.769505
96462   -0.766621
96463   -0.765242
96464   -0.764957
Name: predicted_mean, dtype: float64
```

## Visualizations

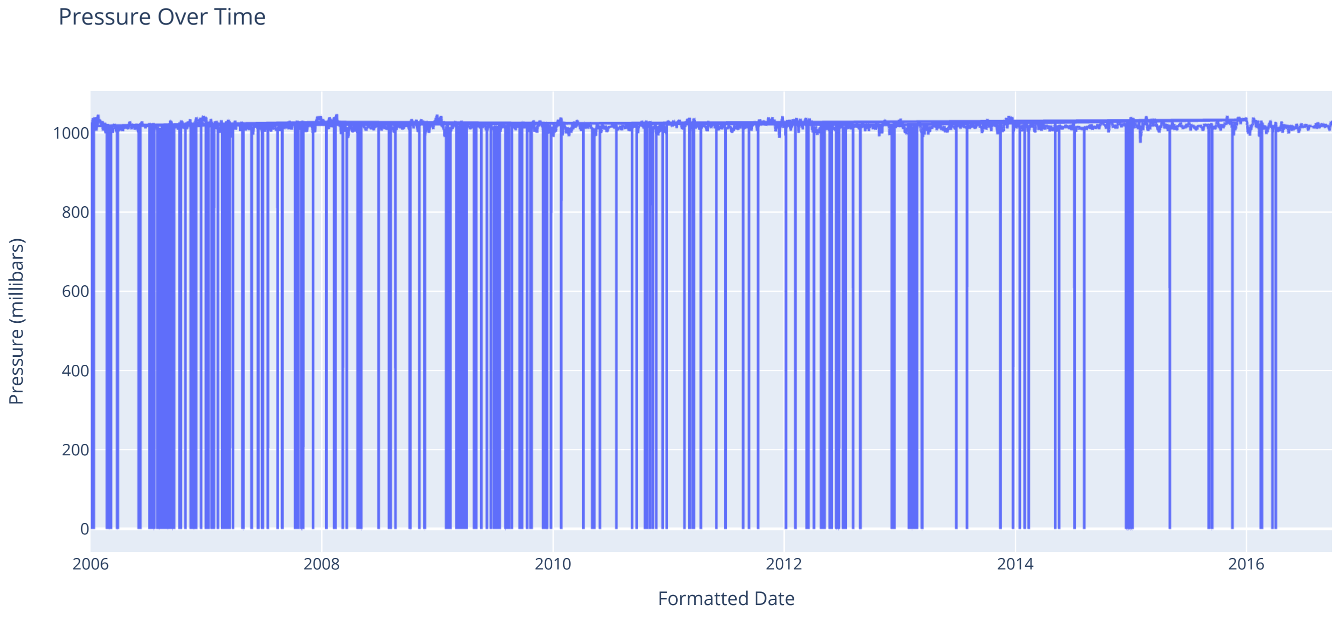
```
import plotly.express as px

# Convert the DataFrame to Pandas (if needed for Plotly)
weather_df_pandas = weather_df.toPandas()

# Create a time series plot of the temperature trends
fig = px.line(weather_df_pandas, x='Formatted Date', y='Temperature (C)', title="Temperature Over Time")
fig.show()
```



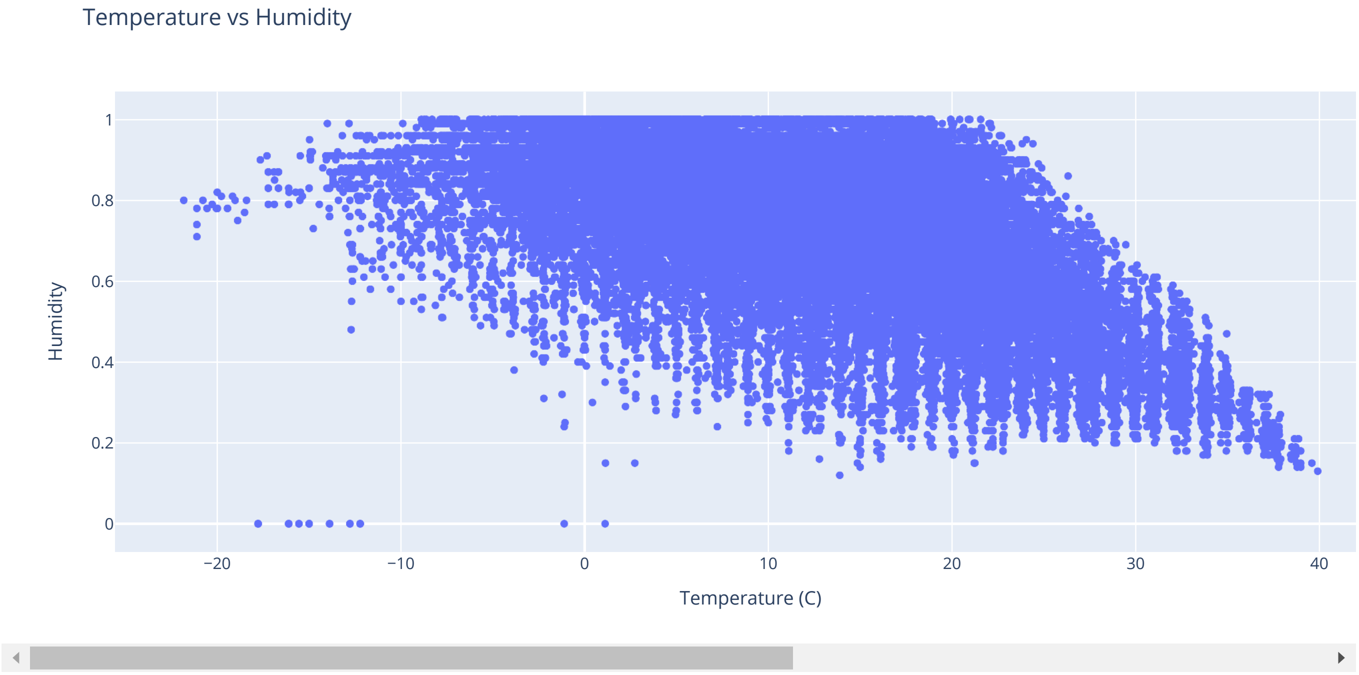
```
# Group by 'Precip Type' and count occurrences
precipitation_trends_pandas = weather_df_pandas['Precip Type'].value_counts().reset_index(name="Count")
precipitation_trends_pandas.columns = ['Precip Type', 'Count']  # Rename columns for clarity

# Now you can create the bar plot
import plotly.express as px

# Bar plot for Precipitation Type Counts
fig_precip = px.bar(precipitation_trends_pandas, x="Precip Type", y="Count", title="Precipitation Type Counts")
fig_precip.show()
```



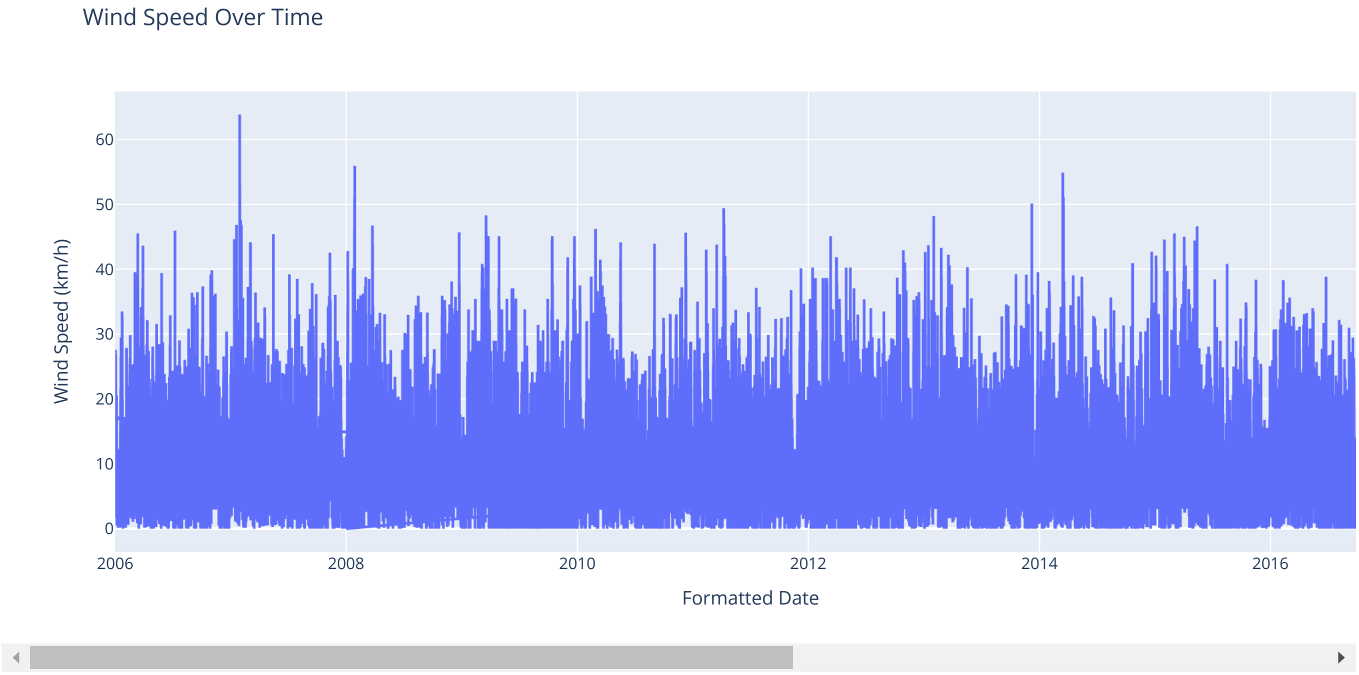
```
# Line plot of Pressure over time
fig_pressure = px.line(weather_df.toPandas(), x="Formatted Date", y="Pressure (millibars)", title="Pressure Over Time")
fig_pressure.show()
```



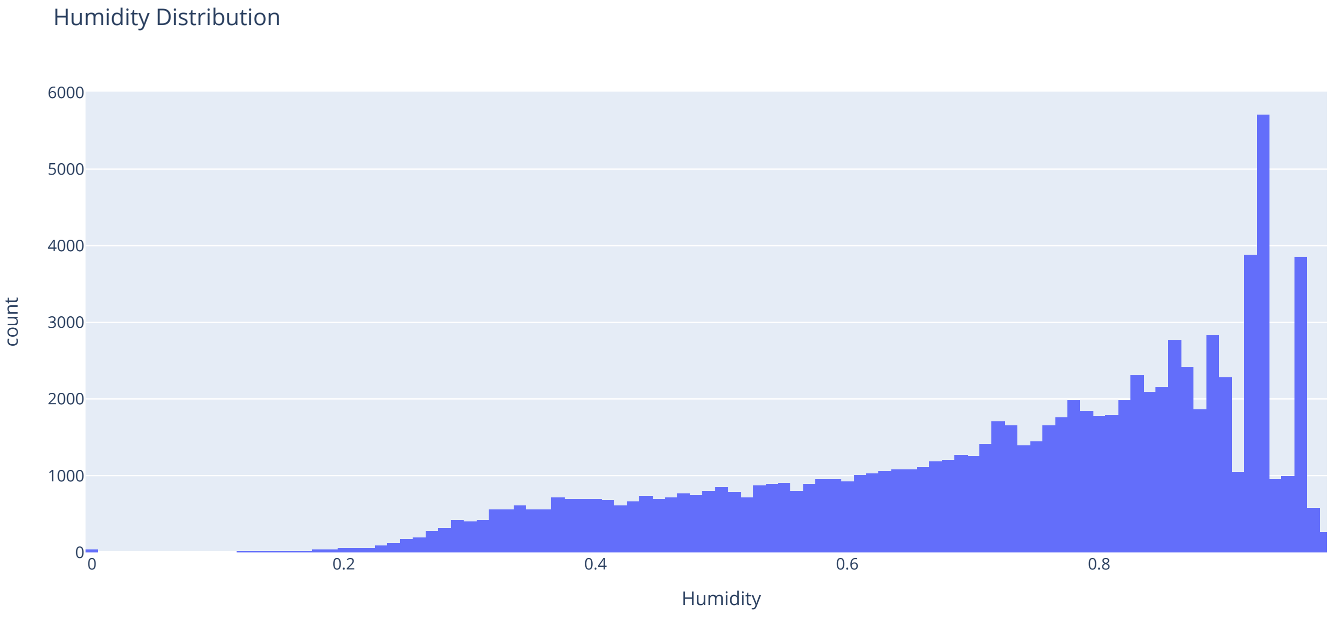
```
# Scatter plot for Temperature vs Humidity
fig_temp_humidity = px.scatter(weather_df.toPandas(), x="Temperature (C)", y="Humidity", title="Temperature vs Humidity")
fig_temp_humidity.show()
```



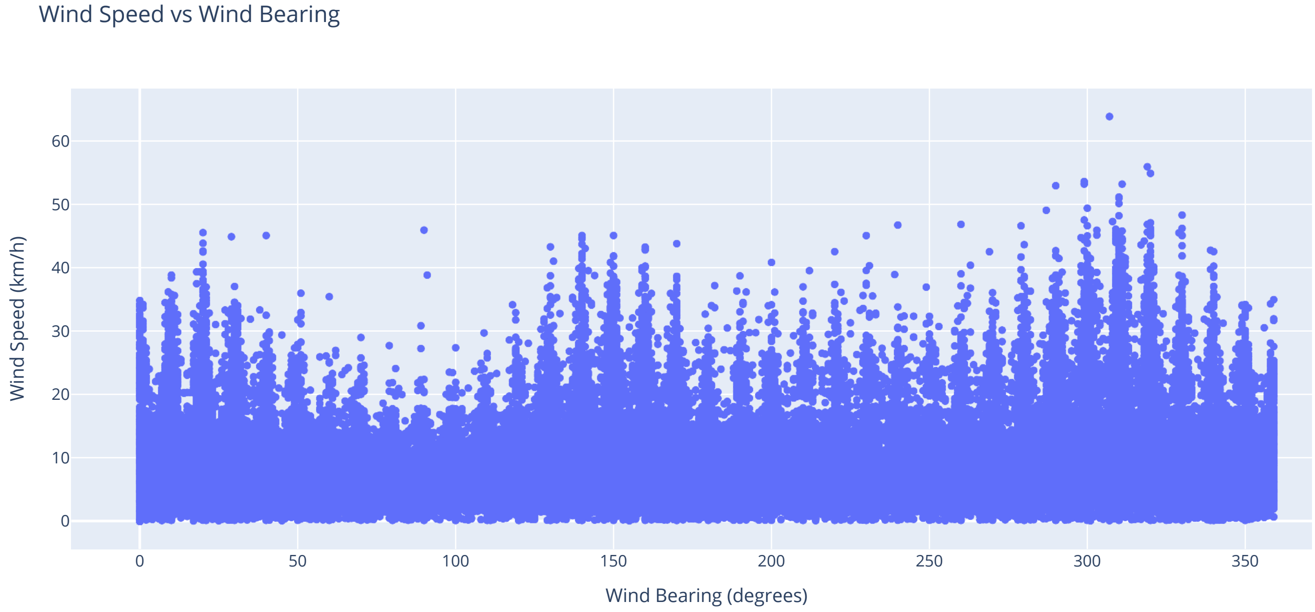
```
# Line plot of Wind Speed over time
fig_wind_speed = px.line(weather_df.toPandas(), x="Formatted Date", y="Wind Speed (km/h)", title="Wind Speed Over Time")
fig_wind_speed.show()
```



```
# Histogram of Humidity
fig_humidity_dist = px.histogram(weather_df.toPandas(), x="Humidity", title="Humidity Distribution")
fig_humidity_dist.show()
```



```
# Scatter plot for Wind Speed vs Wind Bearing
fig_wind_dir = px.scatter(weather_df.toPandas(), x="Wind Bearing (degrees)", y="Wind Speed (km/h)", title="Wind Speed vs Wind Bearing")
fig_wind_dir.show()
```



```
# Count occurrences of different precipitation types
precipitation_trends = weather_df.groupBy("Precip Type").count()

# Show the precipitation type counts
precipitation_trends.show()
```

```
+-----------+-----+
|Precip Type|count|
+-----------+-----+
|       rain|85224|
|       snow|10712|
|       null|  517|
+-----------+-----+
```

## Modelling

## Linear Regression

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator

# Reload the dataset with schema inference
df = spark.read.format("csv") \
  .option("inferSchema", "true") \
  .option("header", "true") \
  .option("sep", ",") \
  .load("weatherHistory.csv")

# Verify column names
print(df.columns)

# Select numeric columns
from pyspark.sql.types import NumericType
numeric_columns = [field.name for field in df.schema.fields if isinstance(field.dataType, NumericType)]
if "Apparent Temperature (C)" not in numeric_columns:
    print("Target column 'Apparent Temperature (C)' not found.")

# Assemble features
feature_columns = [col for col in numeric_columns if col != "Apparent Temperature (C)"]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(df)

# Select target variable and features
data = data.select("features", "Apparent Temperature (C)")

# Split data
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

# Train Linear Regression model
lr = LinearRegression(featuresCol="features", labelCol="Apparent Temperature (C)")
lr_model = lr.fit(train_data)

# Make predictions
predictions = lr_model.transform(test_data)

# Evaluate the model
evaluator = RegressionEvaluator(labelCol="Apparent Temperature (C)", predictionCol="prediction", metricName="mse")
mse = evaluator.evaluate(predictions)
r2_evaluator = RegressionEvaluator(labelCol="Apparent Temperature (C)", predictionCol="prediction", metricName="r2")
r2 = r2_evaluator.evaluate(predictions)

# Print evaluation metrics
print(f"Mean Squared Error (MSE): {mse}")
print(f"R-squared (R²): {r2}")

# Show predictions
predictions.select("Apparent Temperature (C)", "prediction", "features").show(10)
```

```
['Formatted Date', 'Summary', 'Precip Type', 'Temperature (C)', 'Apparent Temperature (C)', 'Humidity', 'Wind Speed (km/h)', 'Wind
Mean Squared Error (MSE): 1.1529901361613253
R-squared (R²): 0.9899153337741742
+------------------------+-------------------+--------------------+
|Apparent Temperature (C)|         prediction|            features|
+------------------------+-------------------+--------------------+
|      2.2222222222222223| 0.8887719558827802|(7,[0,1,4],[2.222...|
|                     5.0|  4.025554210247784|(7,[0,1,4],[5.0,0...|
|       7.222222222222222|  6.311040823402758|(7,[0,1,4],[7.222...|
|                    10.0|  9.640801382919857|(7,[0,1,4],[10.0,...|
|      11.161111111111113| 10.962458952397043|(7,[0,1,4],[11.16...|
|      12.750000000000002|  12.57385989833725|(7,[0,1,4],[12.75...|
|       18.83888888888889| 19.334857856189362|(7,[0,1,4],[18.83...|
|                     5.0| 4.2942599808232895|(7,[0,1,6],[5.0,1...|
|      -21.11111111111111|-25.608275834916768|[-21.111111111111...|
|      -21.11111111111111| -25.57203600835671|[-21.111111111111...|
+------------------------+-------------------+--------------------+
only showing top 10 rows
```

## Random Forest Classifier

```
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Index target column
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol="Precip Type", outputCol="label")
data = indexer.fit(df).transform(df)

# Assemble features
assembler = VectorAssembler(inputCols=["Temperature (C)", "Humidity", "Wind Speed (km/h)", "Pressure (millibars)", "Visibility (km)"], o
data = assembler.transform(data).select("features", "label")

# Train-test split
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

# Train classifier
rf = RandomForestClassifier(featuresCol="features", labelCol="label")
rf_model = rf.fit(train_data)

# Make predictions
predictions = rf_model.transform(test_data)

# Evaluate
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print(f"Accuracy: {accuracy}")
```

```
Accuracy: 0.9802856255821174
```

```
df.groupBy("Precip Type").count().show()
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction")
precision = evaluator.evaluate(predictions, {evaluator.metricName: "weightedPrecision"})
recall = evaluator.evaluate(predictions, {evaluator.metricName: "weightedRecall"})
f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})

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

```
+-----------+-----+
|Precip Type|count|
+-----------+-----+
|       rain|85224|
|       snow|10712|
|       null|  517|
+-----------+-----+

Precision: 0.9814321040979948
Recall: 0.9802856255821174
F1-Score: 0.9795003195737573
```

## Logistic Regression

```
from pyspark.sql.functions import when, col
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Load data
file_location = "weatherHistory.csv"
df = spark.read.csv(file_location, header=True, inferSchema=True)

# Create 'Severe' target column based on thresholds
df = df.withColumn("Severe", when((col("Wind Speed (km/h)") > 40) |
                                  (col("Visibility (km)") < 2) |
                                  (col("Pressure (millibars)") < 950), 1).otherwise(0))

# Select features and target
feature_columns = ["Wind Speed (km/h)", "Visibility (km)", "Pressure (millibars)"]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
data = assembler.transform(df).select("features", "Severe")

# Train-test split
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

# Train logistic regression model
lr = LogisticRegression(featuresCol="features", labelCol="Severe")
```

```
lr_model = lr.fit(train_data)

# Make predictions
predictions = lr_model.transform(test_data)

# Evaluate the model
evaluator = BinaryClassificationEvaluator(labelCol="Severe", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
roc_auc = evaluator.evaluate(predictions)

# Print metrics
print(f"Area Under ROC: {roc_auc}")
predictions.select("Severe", "prediction", "probability").show(10)
```

```
Area Under ROC: 0.9905458203199211
+------+----------+--------------------+
|Severe|prediction|         probability|
+------+----------+--------------------+
|     1|       1.0|[0.05279940648665...|
|     1|       1.0|[0.05887014252306...|
|     1|       1.0|[0.06233441524000...|
|     1|       1.0|[0.05606631956931...|
|     1|       1.0|[0.06316656352092...|
|     1|       1.0|[0.06801024665090...|
|     1|       1.0|[0.07289510257231...|
|     1|       1.0|[0.07569013130023...|
|     1|       1.0|[0.08188071052876...|
|     1|       1.0|[0.08203285851481...|
+------+----------+--------------------+
only showing top 10 rows
```

```
# Get feature importances
print("Feature Importances:")
for col, importance in zip(["Wind Speed", "Visibility", "Pressure", "Cloud Cover"], rf_model.featureImportances):
    print(f"{col}: {importance}")
```

```
Feature Importances:
Wind Speed: 0.9218145920368948
Visibility: 0.009149350080915932
Pressure: 0.0009058513976209076
Cloud Cover: 0.028452587260134866
```

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator, MulticlassClassificationEvaluator

# Evaluate accuracy, precision, recall, F1-score

# Accuracy
evaluator = MulticlassClassificationEvaluator(labelCol="Severe", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)

# Precision
precision_evaluator = MulticlassClassificationEvaluator(labelCol="Severe", predictionCol="prediction", metricName="weightedPrecision")
precision = precision_evaluator.evaluate(predictions)

# Recall
recall_evaluator = MulticlassClassificationEvaluator(labelCol="Severe", predictionCol="prediction", metricName="weightedRecall")
recall = recall_evaluator.evaluate(predictions)

# F1-Score
f1_evaluator = MulticlassClassificationEvaluator(labelCol="Severe", predictionCol="prediction", metricName="f1")
f1 = f1_evaluator.evaluate(predictions)

# Area Under ROC (for binary classification)
binary_evaluator = BinaryClassificationEvaluator(labelCol="Severe", rawPredictionCol="rawPrediction", metricName="areaUnderROC")
roc_auc = binary_evaluator.evaluate(predictions)

# Print metrics
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
print(f"Area Under ROC: {roc_auc}")

# Confusion Matrix
confusion_matrix = predictions.groupBy("Severe", "prediction").count()
confusion_matrix.show()
```

```
Accuracy: 0.9926006416226845
Precision: 0.9925637434613996
Recall: 0.9926006416226845
```

```
F1-Score: 0.9925798077955317
Area Under ROC: 0.9905480808235505
+------+----------+-----+
|Severe|prediction|count|
+------+----------+-----+
|     1|       0.0|   79|
|     0|       0.0|18014|
|     1|       1.0| 1169|
|     0|       1.0|   64|
+------+----------+-----+
```

```
from pyspark.sql.functions import col

# Confusion Matrix: Group by actual label and prediction
confusion_matrix = predictions.groupBy("Severe", "prediction").count()

# Display the Confusion Matrix
print("Confusion Matrix:")
confusion_matrix.show()

# Convert the Confusion Matrix into a readable format (optional)
labels = predictions.select("Severe").distinct().orderBy("Severe").rdd.flatMap(lambda x: x).collect()

matrix = confusion_matrix.collect()

# Initialize an empty dictionary to store confusion matrix
matrix_dict = {label: {label: 0 for label in labels} for label in labels}

for row in matrix:
    actual = row["Severe"]
    predicted = row["prediction"]
    count = row["count"]
    matrix_dict[actual][predicted] = count

# Print Confusion Matrix in readable format
print("\nConfusion Matrix (Readable Format):")
print("Actual \\ Predicted", "\t".join([f"{label}" for label in labels]))
for actual in labels:
```