PROBLEM STATEMENT

TNSDC NAAN MUDHALVAN (IBM)

Project Name: Future Sales Prediction

College code: 1108

College Name: Jaya Engineering College

‘

Team Members:

1. **Thirukarthikeyan (B.Tech[Information Technology])**
2. **Veluprasath (B.Tech[Information Technology])**
3. **Praveen kumar (B.Tech[Information Technology])**
4. **Vignesh (B.Tech[Information Technology])**
5. **Abinesh(B.Tech[Information Technology])**

Introduction:

In the analysis of the "Future Sales Prediction" dataset, we conducted a comprehensive series of data analysis steps to create an accurate prediction model. The process began with Exploratory Data Analysis (EDA) to understand the dataset's characteristics. Subsequently, we performed data preprocessing, including outlier detection and handling using the winzoring technique, as well as data normalization using the min-max method. We then developed multiple models, including Linear Regression, Ridge Regression, Lasso Regression, Decision Tree, and Random Forest, all of which were evaluated through cross-validation. The model evaluation results revealed that Random Forest outperformed others, yielding an average Mean Squared Error (MSE) of 10.32%, Root Mean Squared Error (RMSE) of 8.09%, Mean Absolute Error (MAE) of 5.99%, and an R-squared value of 94.27%. Additionally, we conducted classic assumption tests, including tests for linearity, homoscedasticity, normality, multicollinearity, outliers, and independence, to ensure the validity of our model. These results provide in-depth insights into the quality of our prediction model and its relevance in the context of future sales forecasting.

Load Data:

```
        import pandas as pd

df = pd.read_csv('/kaggle/input/future-sales-prediction/Sales.csv')
df.head()
```

output:

TV	Radio	Newspaper	Sales	
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	12.0
3	151.5	41.3	58.5	16.5
4	180.8	10.8	58.4	17.9

Features explanation:

- **TV**: this feature represents the amount of advertising budget spent on television media for a product or service in a certain period, for example in thousands of dollars (USD).
- **Radio**: this feature represents the amount of advertising budget spent on radio media in the same period as TV.
- **Newspaper**: this feature represents the amount of advertising budget spent in newspapers or print media in the same period as TV and Radio.
- **Sales**: This feature represents product or service sales data in the same period as advertising expenditure on TV, Radio and Newspaper.

Shaping:

```
df.shape
```

Output:

```
(200, 4)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 200 entries, 0 to 199
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype  
---  ------     --------------  -----  
 0   TV         200 non-null    float64
 1   Radio      200 non-null    float64
 2   Newspaper  200 non-null    float64
 3   Sales      200 non-null    float64
dtypes: float64(4)
memory usage: 6.4 KB
```

```
df.describe().T
```

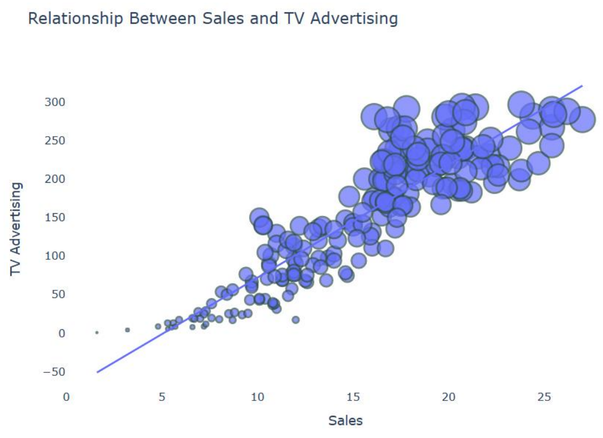
Output:

	count	mean	std	min	25%	50%	75%	max
Tv	200.0	147.0425	85.854236	0.7	74.375	149.75	218.825	296.4
Radio	200.0	23.2640	14.846809	0.0	9.975	22.90	36.525	49.6
News	200.0	30.5540	21.778621	0.3	12.750	25.75	45.100	114.0
Sales	200.0	15.1305	5.283892	1.6	11.000	16.00	19.050	27.0

Exploratory Data Analysis (EDA)

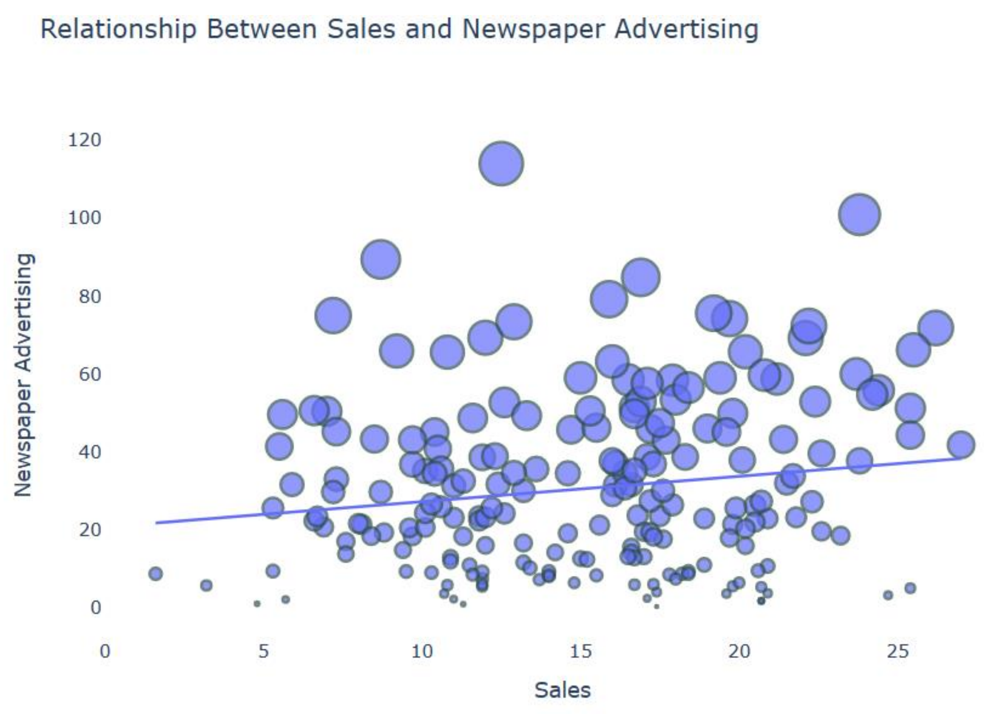
```
import plotly.express as px

figure = px.scatter(df, x='Sales', y='TV', size='TV', trendline='ols',
title='Relationship Between Sales and TV Advertising')
figure.update_traces(marker=dict(line=dict(width=2, color='DarkSlateGre
y')), selector=dict(mode='markers'))
figure.update_layout(
    xaxis_title='Sales',
    yaxis_title='TV Advertising',
    legend_title='TV Ad Size',
    plot_bgcolor='white'
)
figure.show()
```

Output:



```
figure = px.scatter(df, x='Sales', y='Newspaper', size='Newspaper', tre
ndline='ols', title='Relationship Between Sales and Newspaper Advertisi
ng')
figure.update_traces(marker=dict(line=dict(width=2, color='DarkSlateGre
y')), selector=dict(mode='markers'))
figure.update_layout(
    xaxis_title='Sales',
    yaxis_title='Newspaper Advertising',
    legend_title='Newspaper Ad Size',
    plot_bgcolor='white'
)
figure.show()
```

Output:



```
# Calculate the correlation
correlation = df.corr()
sales_correlation = correlation["Sales"].sort_values(ascending=False)

# Format and style the correlation values
styled_sales_correlation = sales_correlation.apply(lambda x: f'{x:.2f}'
)
styled_sales_correlation = styled_sales_correlation.reset_index()
styled_sales_correlation.columns = ["Feature", "Correlation with Sales"
]
styled_sales_correlation.style.background_gradient(cmap='coolwarm', axi
s=0)
```

Output:

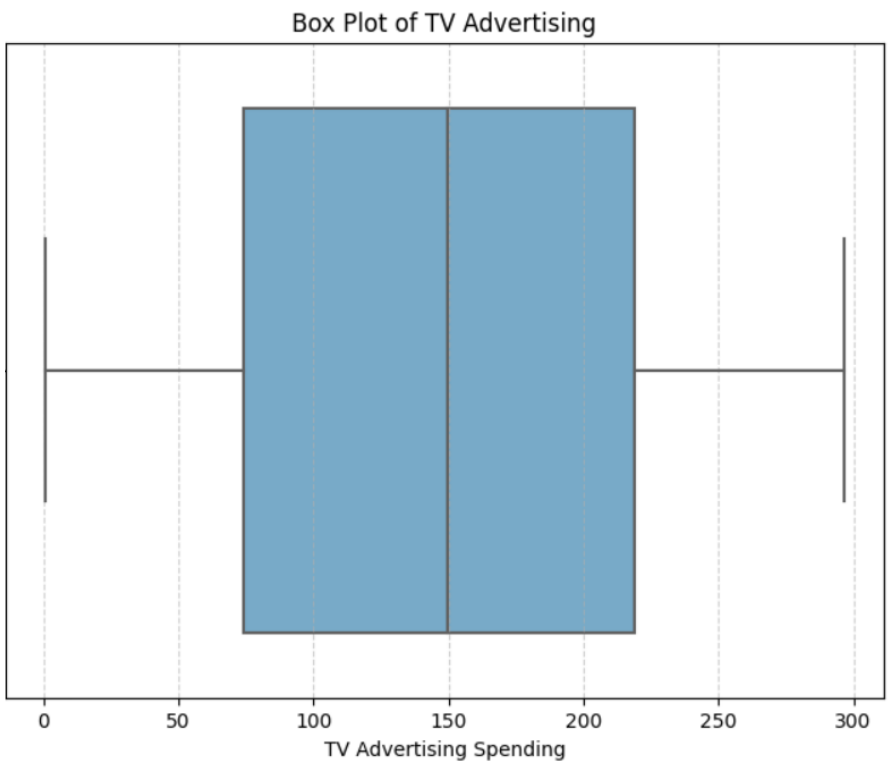
	Feature	Correlation with Sales
0	Sales	1.00
1	TV	0.90
2	Radio	0.35
3	Newspaper	0.16

Data Preprocessing:

Outlier detection

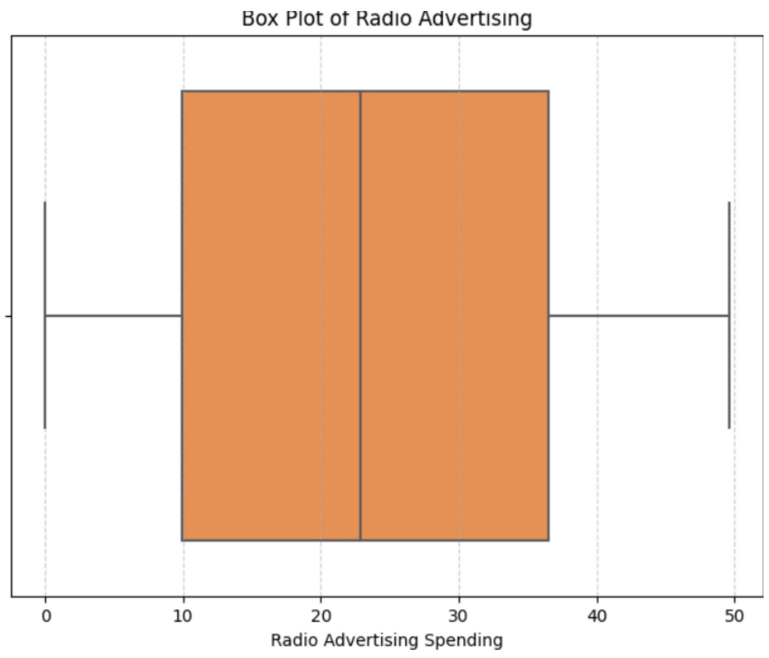
```
import seaborn as sns
import matplotlib.pyplot as plt

# Create the box plot
plt.figure(figsize=(8, 6))
sns.boxplot(x='TV', data=df, palette='Blues')
plt.title('Box Plot of TV Advertising')
plt.xlabel('TV Advertising Spending')
plt.grid(axis='x', linestyle='--', alpha=0.6)

# Show the plot
plt.show()
```

Output:



```
# Create the box plot
plt.figure(figsize=(8, 6))
sns.boxplot(x='Radio', data=df, palette='Oranges')
plt.title('Box Plot of Radio Advertising')
plt.xlabel('Radio Advertising Spending')
plt.grid(axis='x', linestyle='--', alpha=0.6)

# Show the plot
plt.show()
```

Output:



Data normalization:

At this stage, we use the min-max technique. Min-Max is a data preprocessing technique used in data analysis and machine learning to convert values in a dataset into a certain range, usually between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler

# Create a MinMaxScaler object
scaler = MinMaxScaler()

# Columns to be normalized (e.g., TV, Radio, Newspaper)
columns_to_normalize = ['TV', 'Radio', 'Newspaper']

# Apply Min-Max normalization to the selected columns
df[columns_to_normalize] = scaler.fit_transform(df[columns_to_normalize
])
df.head()
```

Output:

	TV	**Radio**	**Newspaper**	**Sales**	**TV**
0	0.775786	0.762097	0.934843	22.1	0
1	0.148123	0.792339	0.607851	10.4	1
2	0.055800	0.925403	0.936200	12.0	2
3	0.509976	0.832661	0.789664	16.5	3
4	0.609063	0.217742	0.788307	17.9	4

Modelling and Evaluation:

At the modeling stage, we use 5 algorithms for comparison, namely Linear Regression, Ridge Regression, Lasso Regression, Decision Tree, and Random Forest.

And for evaluation using MSE, RMSE, MAE and R-Squared.

```
X = df[['TV', 'Radio', 'Newspaper']]
y = df['Sales']
```

```
from sklearn.model_selection import cross_val_score

# Performing 5-fold cross-validation (can be adjusted to the desired numb
er of folds)
num_folds = 5

# Function to perform cross-validation and calculate metrics in percentag
e
def perform_cross_validation(model, X, y, num_folds):
    mse_scores = -cross_val_score(model, X, y, cv=num_folds, scoring='n
eg_mean_squared_error')
    rmse_scores = np.sqrt(mse_scores)
    mae_scores = -cross_val_score(model, X, y, cv=num_folds, scoring='n
eg_mean_absolute_error')
    r2_scores = cross_val_score(model, X, y, cv=num_folds, scoring='r2'
)

    return mse_scores, rmse_scores, mae_scores, r2_scores
```

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso

# Linear Regression
linear_model = LinearRegression()
linear_mse, linear_rmse, linear_mae, linear_r2 = perform_cross_validati
on(linear_model, X, y, num_folds)
print("Linear Regression:")
print(f"Average MSE: {np.mean(linear_mse) / np.mean(y) * 100:.2f}%")
print(f"Average RMSE: {np.mean(linear_rmse) / np.mean(y) * 100:.2f}%")
print(f"Average MAE: {np.mean(linear_mae) / np.mean(y) * 100:.2f}%")
print(f"Average R-squared: {np.mean(linear_r2) * 100:.2f}%")
print("\n")
```

Output:

```
Linear Regression:
Average MSE: 18.90%
Average RMSE: 11.01%
Average MAE: 8.38%
Average R-squared: 89.53%
```

```
# Ridge Regression
ridge_model = Ridge(alpha=1.0)  # You can adjust alpha as needed
ridge_mse, ridge_rmse, ridge_mae, ridge_r2 = perform_cross_validation(r
idge_model, X, y, num_folds)
print("Ridge Regression:")
print(f"Average MSE: {np.mean(ridge_mse) / np.mean(y) * 100:.2f}%")
print(f"Average RMSE: {np.mean(ridge_rmse) / np.mean(y) * 100:.2f}%")
print(f"Average MAE: {np.mean(ridge_mae) / np.mean(y) * 100:.2f}%")
print(f"Average R-squared: {np.mean(ridge_r2) * 100:.2f}%")
print("\n")
```

Output:

```
Ridge Regression:
Average MSE: 19.67%
Average RMSE: 11.20%
Average MAE: 8.54%
Average R-squared: 89.19%
```

```
# Lasso Regression
lasso_model = Lasso(alpha=1.0)  # You can adjust alpha as needed
lasso_mse, lasso_rmse, lasso_mae, lasso_r2 = perform_cross_validation(l
asso_model, X, y, num_folds)
print("Lasso Regression:")
print(f"Average MSE: {np.mean(lasso_mse) / np.mean(y) * 100:.2f}%")
print(f"Average RMSE: {np.mean(lasso_rmse) / np.mean(y) * 100:.2f}%")
print(f"Average MAE: {np.mean(lasso_mae) / np.mean(y) * 100:.2f}%")
print(f"Average R-squared: {np.mean(lasso_r2) * 100:.2f}%")
print("\n")
```

```
Lasso Regression:
Average MSE: 115.55%
Average RMSE: 27.51%
Average MAE: 22.39%
Average R-squared: 35.98%
```

Classic assumption test:

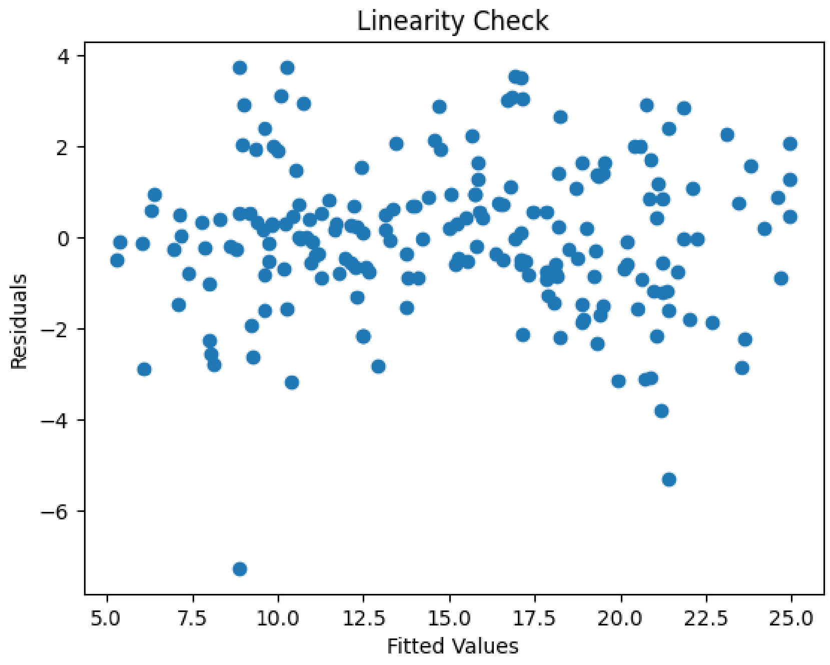
At the classical assumption testing stage, 5 assumption tests are used, namely linearity test, homoscedasticity test, normality test, multicollinearity test, outliers test, and independent test.

```
import statsmodels.api as sm
import statsmodels.stats.api as sms

# Adding a constant to the independent variables (intercept)
X = sm.add_constant(X)
```

```python
# Fit the regression model
model = sm.OLS(y, X).fit()

# Residuals (model residuals)
residuals = model.resid
```

```python
Assumption 1: Linearity
# You can check linearity using residual vs. fitted values plot
import matplotlib.pyplot as plt
plt.scatter(model.fittedvalues, residuals)
plt.xlabel("Fitted Values")
plt.ylabel("Residuals")
plt.title("Linearity Check")
plt.show()
```

Output:



```python
# Assumption 2: Homoskedasticity
# You can check homoskedasticity using Breusch-Pagan test
_, p_homo, _, _ = sms.het_breuschpagan(residuals, X)
print(f"Homoskedasticity (Breusch-Pagan): p-value = {p_homo:.4f}")
```

Output:

```
Homoskedasticity (Breusch-Pagan): p-value = 0.2634
```

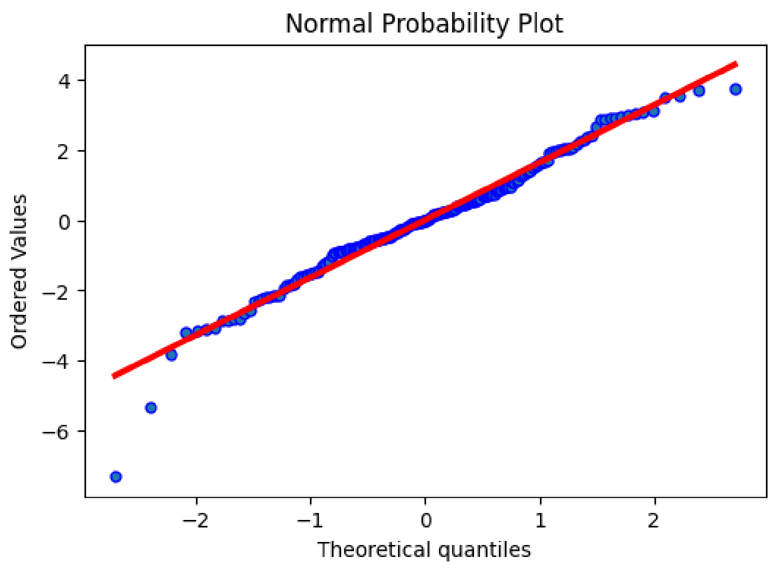
```
# Assumption 3: Independence (Serial Correlation)
# You can check for serial correlation using Durbin-Watson test
from statsmodels.stats.stattools import durbin_watson
dw_stat = durbin_watson(residuals)
print(f"Serial Correlation (Durbin-Watson): DW Statistic = {dw_stat:.2f
}")
```

Output:

```
Serial Correlation (Durbin-Watson): DW Statistic = 2.25
```

```
# Assumption 4: Normality
# You can check normality using a normal probability plot (Q-Q plot)
import scipy.stats as stats
fig, ax = plt.subplots(figsize=(6, 4))
_, (__, ___, r) = stats.probplot(residuals, plot=ax, fit=True)
ax.get_lines()[0].set_markerfacecolor('C0')
ax.get_lines()[0].set_markersize(5.0)
ax.get_lines()[1].set_linewidth(3.0)
plt.title("Normal Probability Plot")
plt.show()
```

Output:



```
# Assumption 5: Multicollinearity
# You can check multicollinearity using the Variance Inflation Factor (VI
F)
from statsmodels.stats.outliers_influence import variance_inflation_fac
tor
vif = pd.DataFrame()
vif["Features"] = X.columns
vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.s
hape[1])]
```

```
print("Multicollinearity (VIF):")
print(vif)
```

Output:

```
Multicollinearity (VIF):
    Features       VIF
0      const  6.898975
1         TV  1.005037
2      Radio  1.150018
3  Newspaper  1.150920
```

```
# Assumption 6: Outliers
# You can check for outliers using the studentized residuals and the Cook
's distance
student_resid = model.get_influence().resid_studentized_internal
cooks_d = model.get_influence().cooks_distance[0]
outliers = pd.DataFrame({'Studentized Residuals': student_resid, "Cook'
s Distance": cooks_d})
outliers.index = X.index
print("Outliers:")
print(outliers[outliers['Studentized Residuals'].abs() > 2])  # You can
adjust the threshold as needed
```

```
Outliers:
     Studentized Residuals  Cook's Distance
10                2.272004         0.021004
33               -2.322006         0.037363
97                2.148943         0.007995
130              -4.468814         0.195094
150              -3.233182         0.056641
154               2.120557         0.013399
196               2.261963         0.021244
```

Time Series Forecasting with Prophet:

- Now, we will describe how to use the Prophet library to predict future values of our time series data.
- The developers of Prophet have made it more intuitive for analysts and developers alike to work with time series data.
- To begin, we must instantiate a new Prophet object. Prophet enables us to specify a number of arguments. For example, we can specify the desired range of our uncertainty interval by setting the `interval_width parameter`.

```
# set the uncertainty interval to 95% (the Prophet default is 80%)
```

```
my_model = Prophet(interval_width=0.95)
```

- Now that our Prophet model has been initialized, we can call its `fit` method with our DataFrame as input.

```
my_model.fit(df)
```

Output:

```
<fbprophet.forecaster.Prophet at 0x7f9255c62c90>
```

- In order to obtain forecasts of our time series, we must provide Prophet with a new DataFrame containing a `ds` column that holds the dates for which we want predictions.
- Conveniently, we do not have to concern ourselves with manually creating this DataFrame, as Prophet provides the `make_future_dataframe` helper function.

```
future_dates = my_model.make_future_dataframe(periods=36, freq='MS')
future_dates.head()
```

Output:

	ds
0	1949-01-01
1	1949-02-01
2	1949-03-01
3	1949-04-01
4	1949-05-01

- In the code snippet above, we instructed Prophet to generate 36 datestamps in the future.
- When working with Prophet, it is important to consider the frequency of our time series.
- Because we are working with monthly data, we clearly specified the desired frequency of the timestamps (in this case, `MS` is the start of the month).
- Therefore, the `make_future_dataframe` generated 36 monthly timestamps for us.
- In other words, we are looking to predict future values of our time series 3 years into the future.
- The DataFrame of future dates is then used as input to the predict method of our fitted model.

  Code:

```
forecast = my_model.predict(future_dates)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].head()
```

Output:

	ds	yhat	yhat_lower	yhat_upper
0	1949-01-01	85.667868	40.711662	130.242936
1	1949-02-01	79.176553	35.772575	124.991217
2	1949-03-01	110.839332	69.379890	155.182425
3	1949-04-01	108.472210	67.188442	153.991365
4	1949-05-01	111.854130	69.954869	157.146512

Prophet returns a large DataFrame with many interesting columns, but we subset our output to the columns most relevant to forecasting.

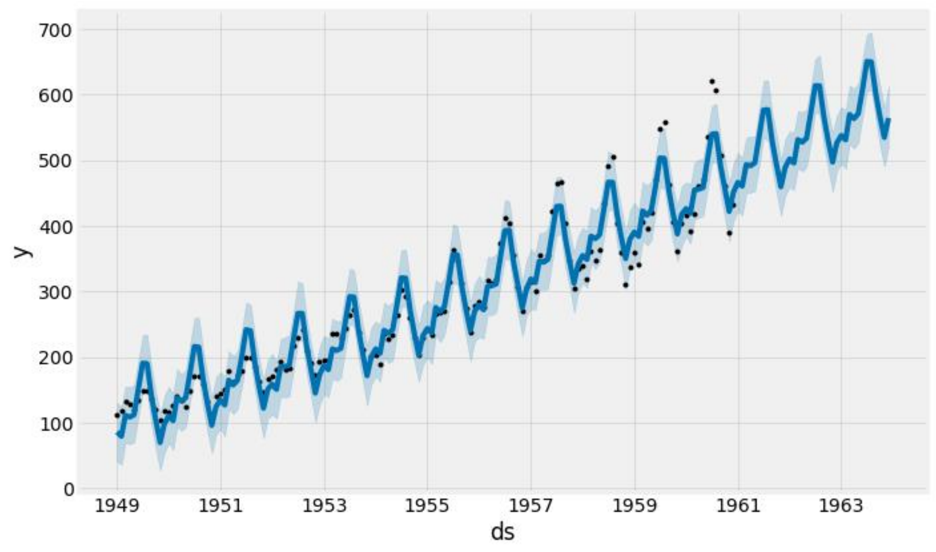
These are:

- **ds**: the datestamp of the forecasted value.
- **yhat**: the forecasted value of our metric (in Statistics, yhat is a notation traditionally used to represent the predicted values of a value y).
- **yhat_lower**: the lower bound of our forecasts.
- **yhat_upper**: the upper bound of our forecasts.
- A variation in values from the output presented is to be expected as Prophet relies on **Markov chain Monte Carlo (MCMC)** methods to generate its forecasts.
- MCMC is a stochastic process, so values will be slightly different each time

- Prophet also provides a convenient function to quickly plot the results of our forecasts as follows:

```
my_model.plot(forecast, uncertainty=True)
```

Output:



Explaination:

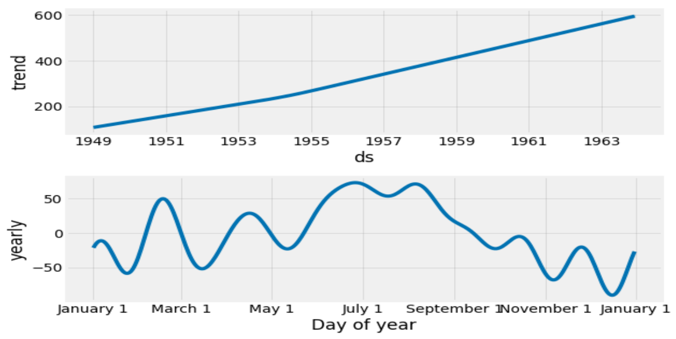
- Prophet plots the observed values of our time series (the black dots), the forecasted values (blue line) and the uncertainty intervals of our forecasts (the blue shaded regions).

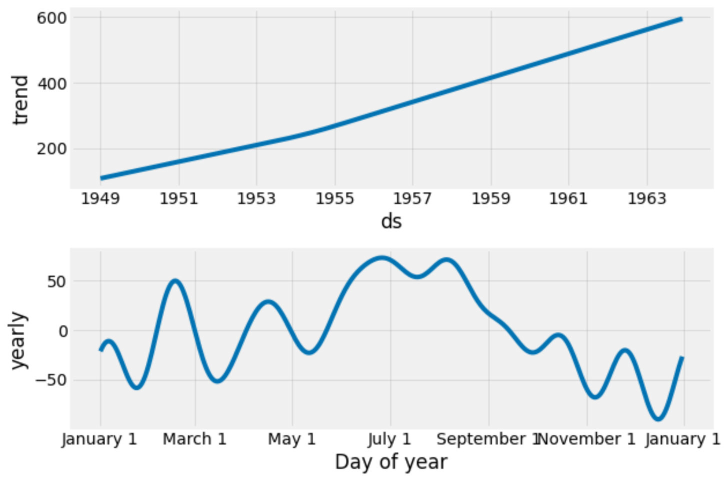
- One other particularly strong feature of Prophet is its ability to return the components of our forecasts.
- This can help reveal how daily, weekly and yearly patterns of the time series contribute to the overall forecasted values

Code:

```
my_model.plot_components(forecast
```

Output:





Explaination:

- The above plot provides interesting insights.
- The first plot shows that the monthly volume of airline passengers has been linearly increasing over time.
- The second plot highlights the fact that the weekly count of passengers peaks towards the end of the week and on Saturday.
- The third plot shows that the most traffic occurs during the holiday months of July and August.

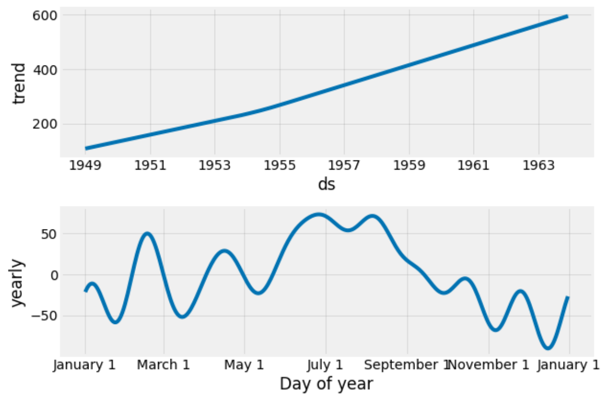
Plotting the forecasted components :

- We can plot the trend and seasonality, components of the forecast as follows:

Code:

```
fig1 = my_model.plot_components(forecast)
```

Output:



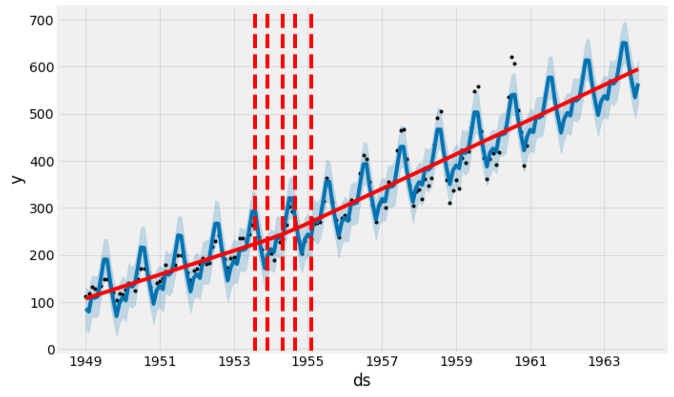
Adding ChangePoints to Prophet :

- Changepoints are the datetime points where the time series have abrupt changes in the trajectory.
- By default, Prophet adds 25 changepoints to the initial 80% of the data-set.
- Let's plot the vertical lines where the potential changepoints occurred.

Code:

```
from fbprophet.plot import add_changepoints_to_plot
fig = my_model.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), my_model, forecast)
```

Output:



We can view the dates where the chagepoints occurred:

Code:

```
my_model.changepoints
```

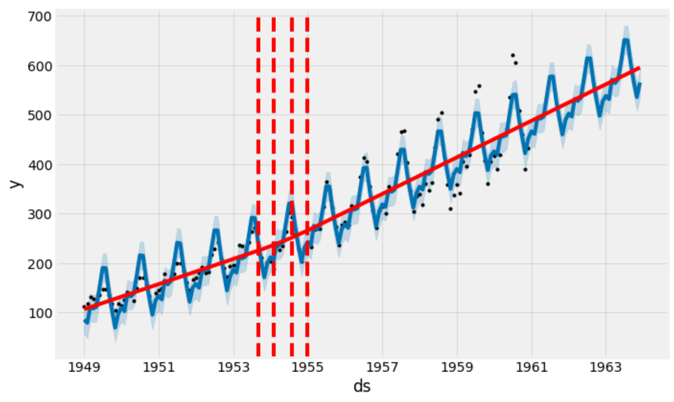
Output:

```
5     1949-06-01
9     1949-10-01
14    1950-03-01
18    1950-07-01
23    1950-12-01
27    1951-04-01
32    1951-09-01
36    1952-01-01
41    1952-06-01
46    1952-11-01
50    1953-03-01
55    1953-08-01
59    1953-12-01
64    1954-05-01
68    1954-09-01
73    1955-02-01
78    1955-07-01
82    1955-11-01
87    1956-04-01
91    1956-08-01
96    1957-01-01
100   1957-05-01
105   1957-10-01
109   1958-02-01
114   1958-07-01
Name: ds, dtype: datetime64[ns]
```

We can change the inferred changepoint range by setting the *changepoint_range*

```
pro_change= Prophet(changepoint_range=0.9)
forecast = pro_change.fit(df).predict(future_dates)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

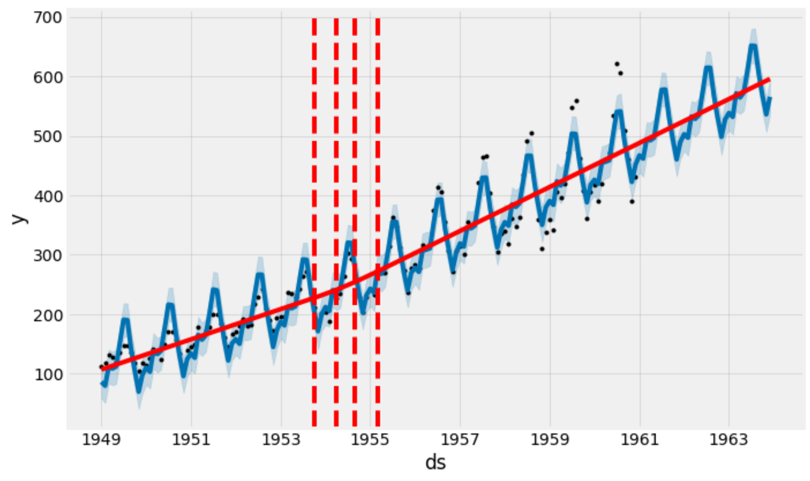
Output:



The number of changepoints can be set by using the *n_changepoints* parameter when initializing prophet.

```
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True)
forecast = pro_change.fit(df).predict(future_dates)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

Output:

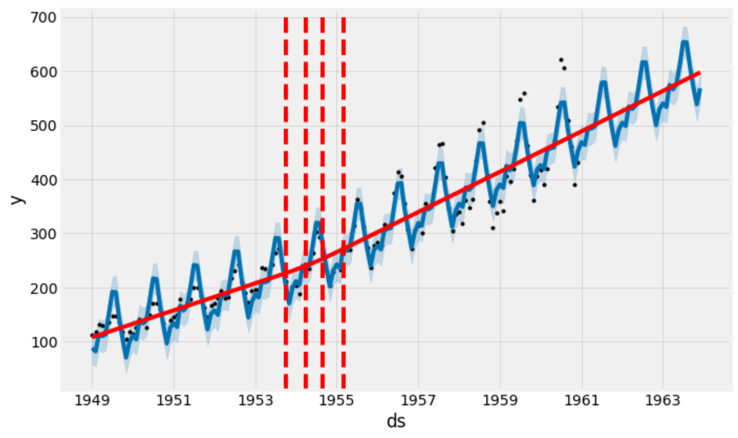


Adjusting Trend:

- Prophet allows us to adjust the trend in case there is an overfit or underfit.
- *changepoint_prior_scale* helps adjust the strength of the trend.
- Default value for *changepoint_prior_scale* is 0.05.
- Decrease the value to make the trend less flexible.
- Increase the value of changepoint_prior_scale to make the trend more flexible.
- Increasing the *changepoint_prior_scale* to 0.08 to make the trend flexible.

```
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True, c
hangepoint_prior_scale=0.08)
forecast = pro_change.fit(df).predict(future_dates)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

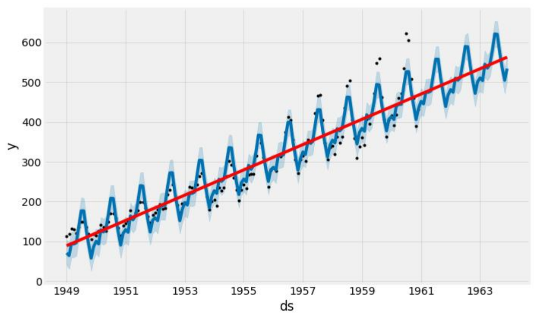
Output:



- Decreasing the *changepoint_prior_scale* to 0.001 to make the trend less flexible.

```
pro_change= Prophet(n_changepoints=20, yearly_seasonality=True, c
hangepoint_prior_scale=0.001)
forecast = pro_change.fit(df).predict(future_dates)
fig= pro_change.plot(forecast);
a = add_changepoints_to_plot(fig.gca(), pro_change, forecast)
```

Output:



Conclusion:

- In this tutorial, we described how to use the Prophet library to perform time series forecasting in Python.
- We have been using out-of-the box parameters, but Prophet enables us to specify many more arguments.
- In particular, Prophet provides the functionality to bring your own knowledge about time series to the table.