RAPPORT DU PROJET:

“Prédiction du Trafic Urbain à l’aide de Données IoT pour une Ville Intelligente”

Phase 1 : Exploration et Compréhension de l’Ensemble de Données

```
# Aperçu des premières lignes de données
print("Aperçu des données :")
data.head()
```

```
Aperçu des données :
```

	DateTime	Junction	Vehicles	ID
0	11/1/2015 0:00	1	15	20151101001
1	11/1/2015 1:00	1	13	20151101011
2	11/1/2015 2:00	1	10	20151101021
3	11/1/2015 3:00	1	7	20151101031
4	11/1/2015 4:00	1	9	20151101041

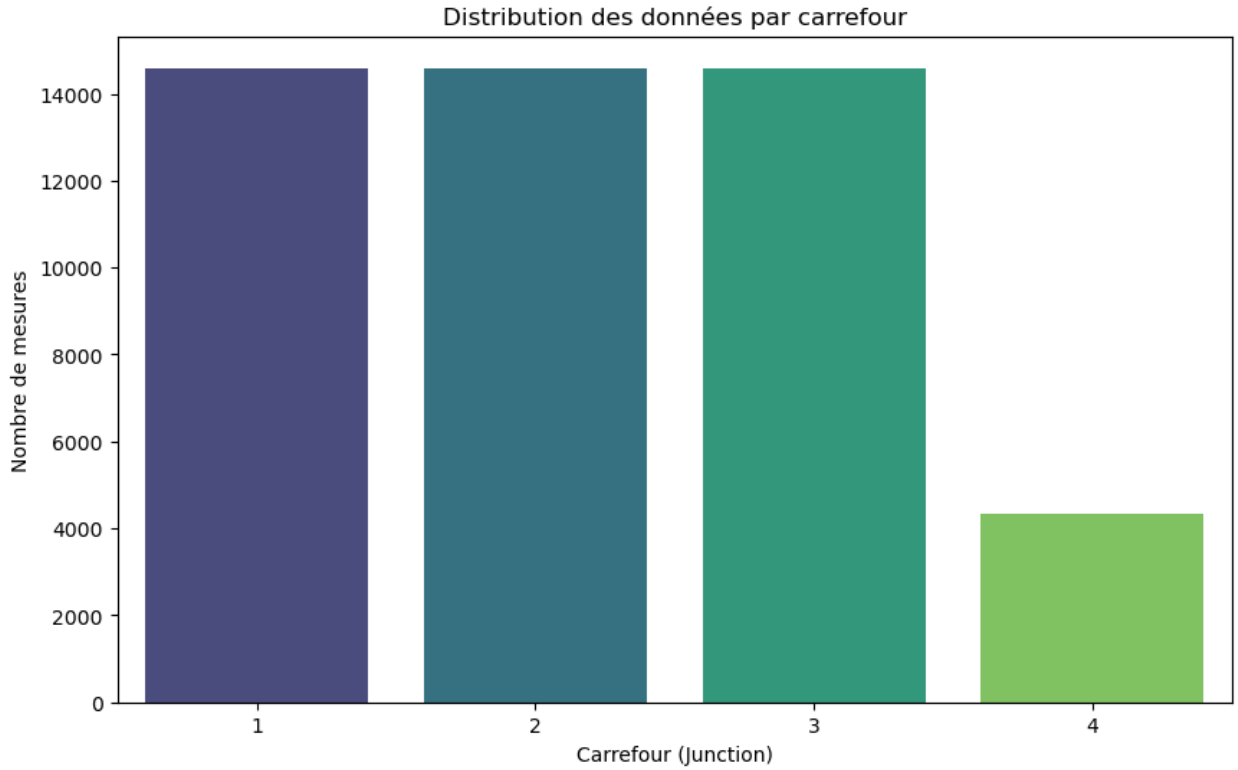
```
# Information générale sur les données (type de chaque colonne, valeurs manquantes)
print("\nInformations générales :")
print(data.info())
```

```
Informations générales :
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 48120 entries, 0 to 48119
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype 
---  ------    --------------  ----- 
 0   DateTime  48120 non-null  object
 1   Junction  48120 non-null  int64 
 2   Vehicles  48120 non-null  int64 
 3   ID        48120 non-null  int64 
dtypes: int64(3), object(1)
memory usage: 1.5+ MB
None
```

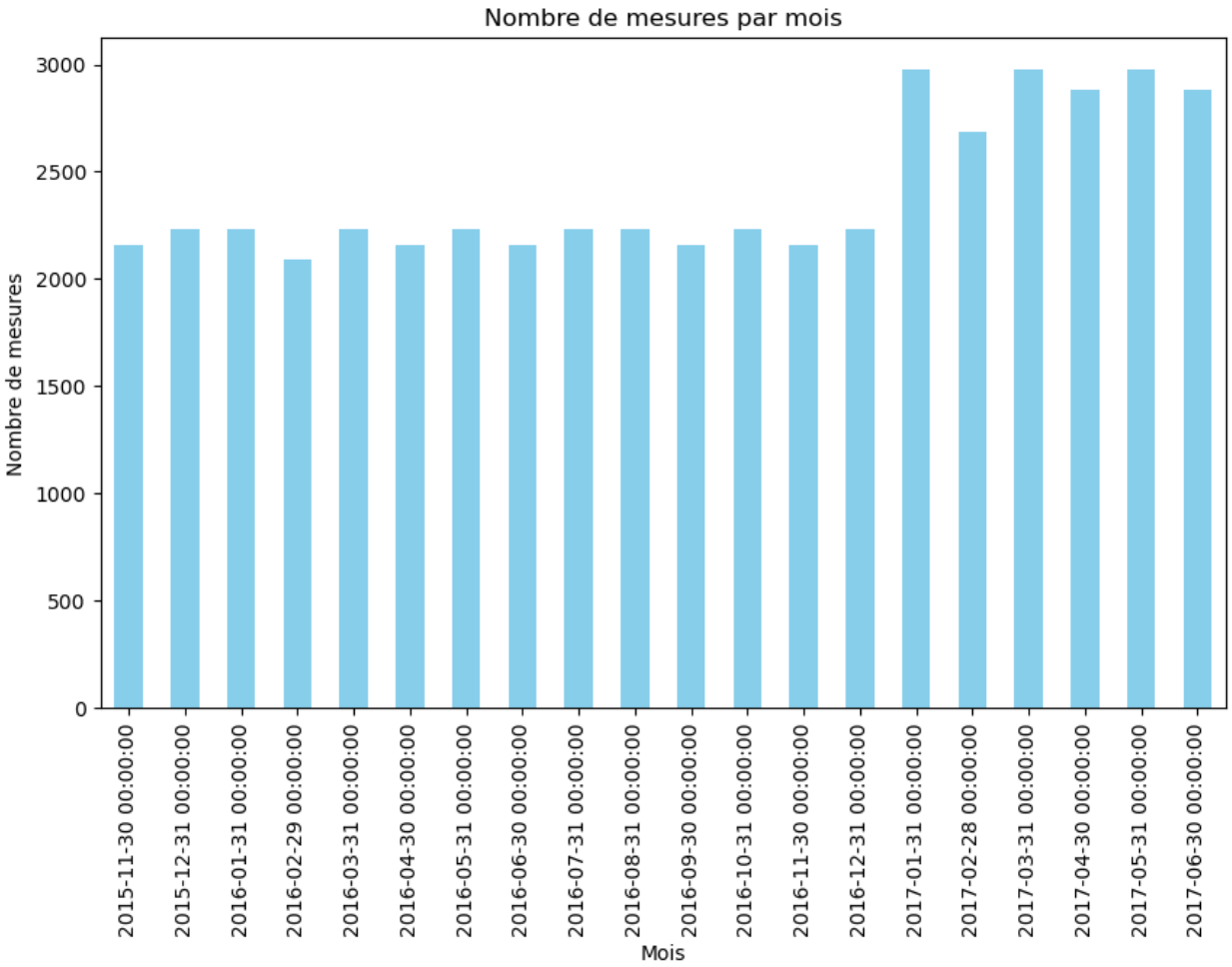
Distribution des données par carrefour (Junction)

```
plt.figure(figsize=(10, 6))
sns.countplot(data=data, x='Junction', palette='viridis')
plt.title("Distribution des données par carrefour")
plt.xlabel("Carrefour (Junction)")
plt.ylabel("Nombre de mesures")
plt.show()
```



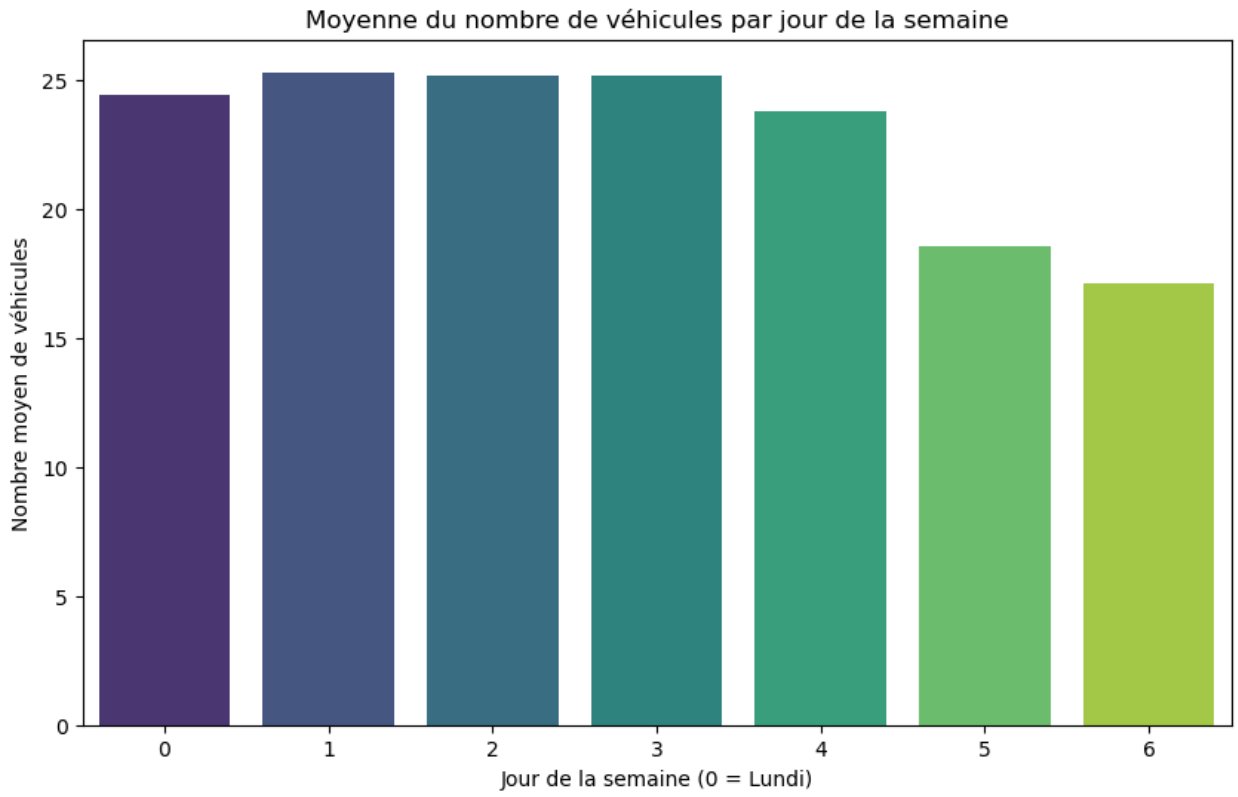
1. Aperçu de la couverture temporelle

```
data['DateTime'] = pd.to_datetime(data['DateTime'])  # Convertir en format datetime
data.set_index('DateTime', inplace=True)  # Définir DateTime comme index
print(f"Plage temporelle des données : {data.index.min()} à {data.index.max()}")

# Vérification de la couverture temporelle (par mois)
coverage = data.resample('M').size()
plt.figure(figsize=(10, 6))
coverage.plot(kind='bar', color='skyblue')
plt.title("Nombre de mesures par mois")
plt.xlabel("Mois")
plt.ylabel("Nombre de mesures")
plt.show()
```



Réponse1.Périodes bien couvertes : janvier, mars ,mai & juin

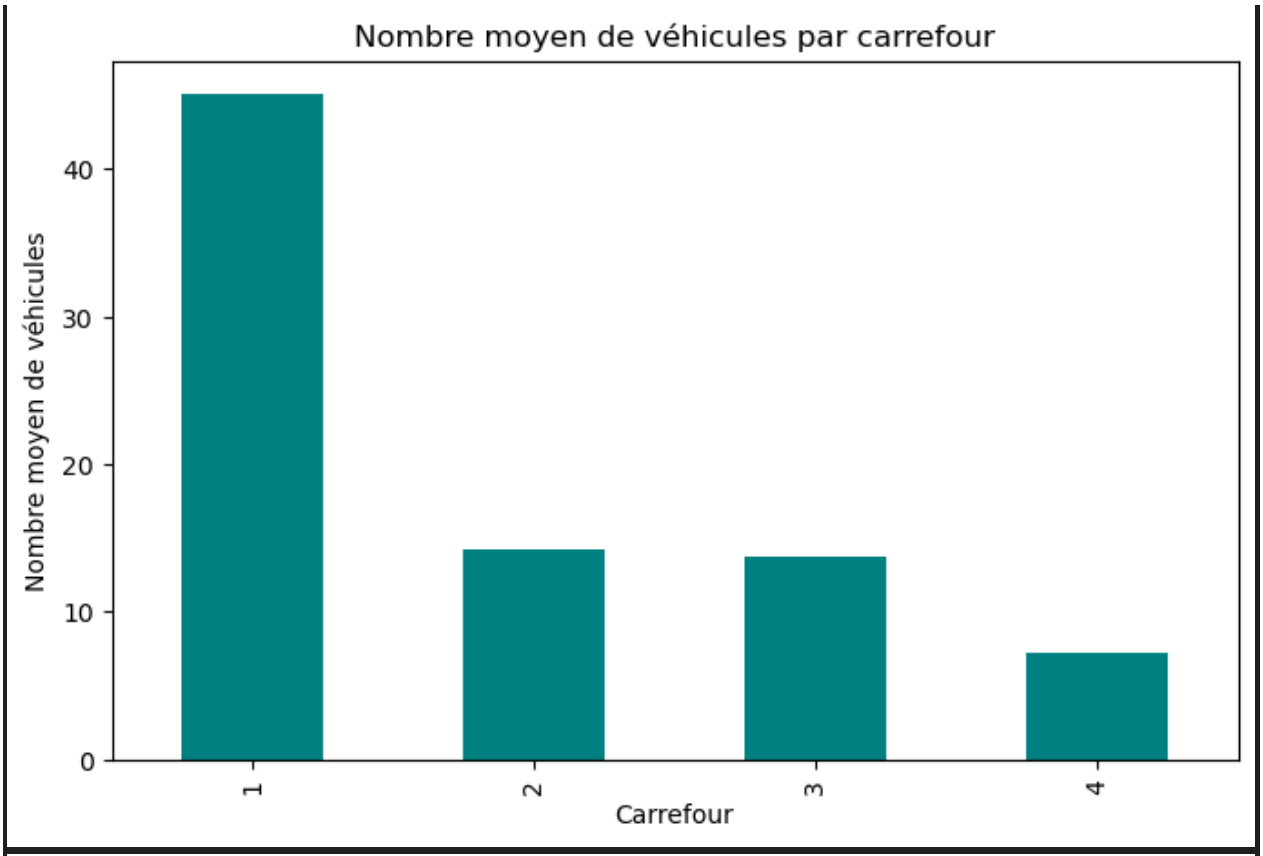
2. Identification des schémas de circulation

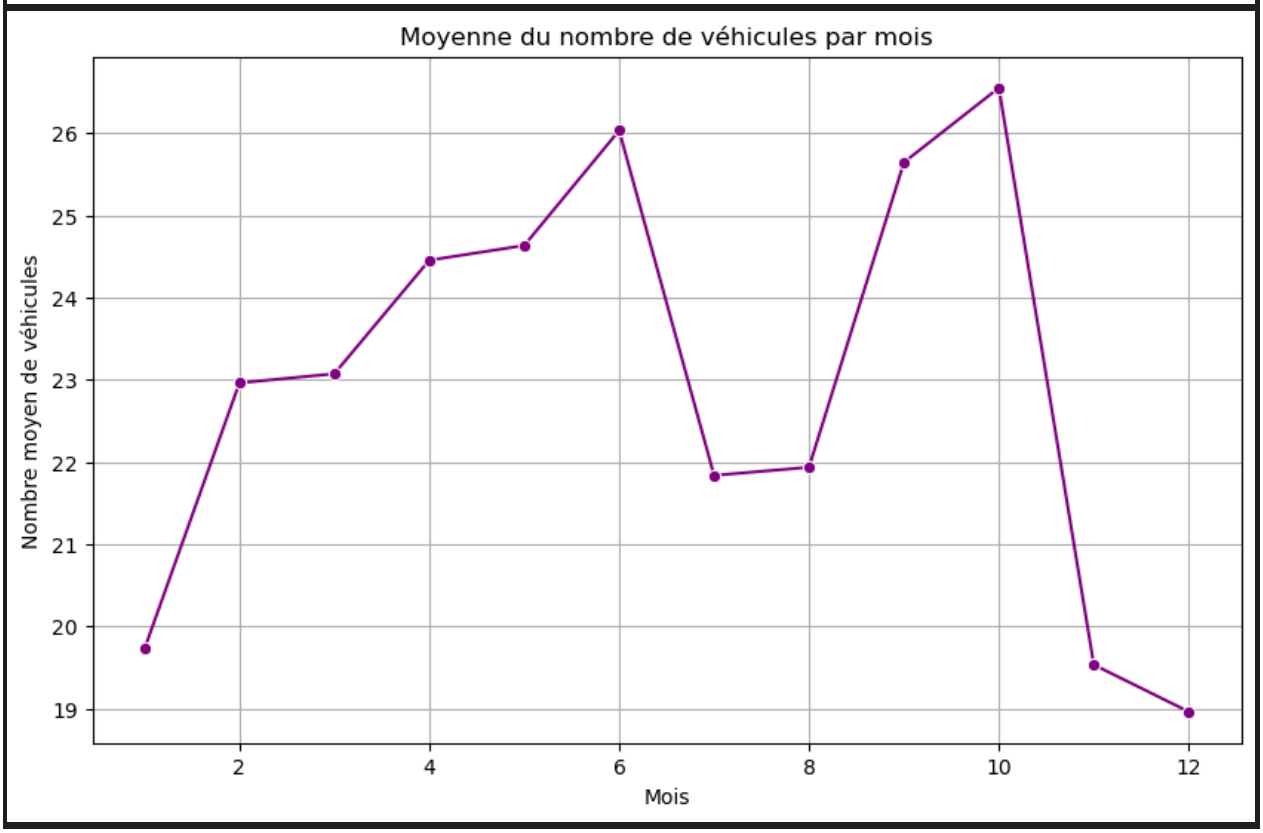


3. Sources de variation

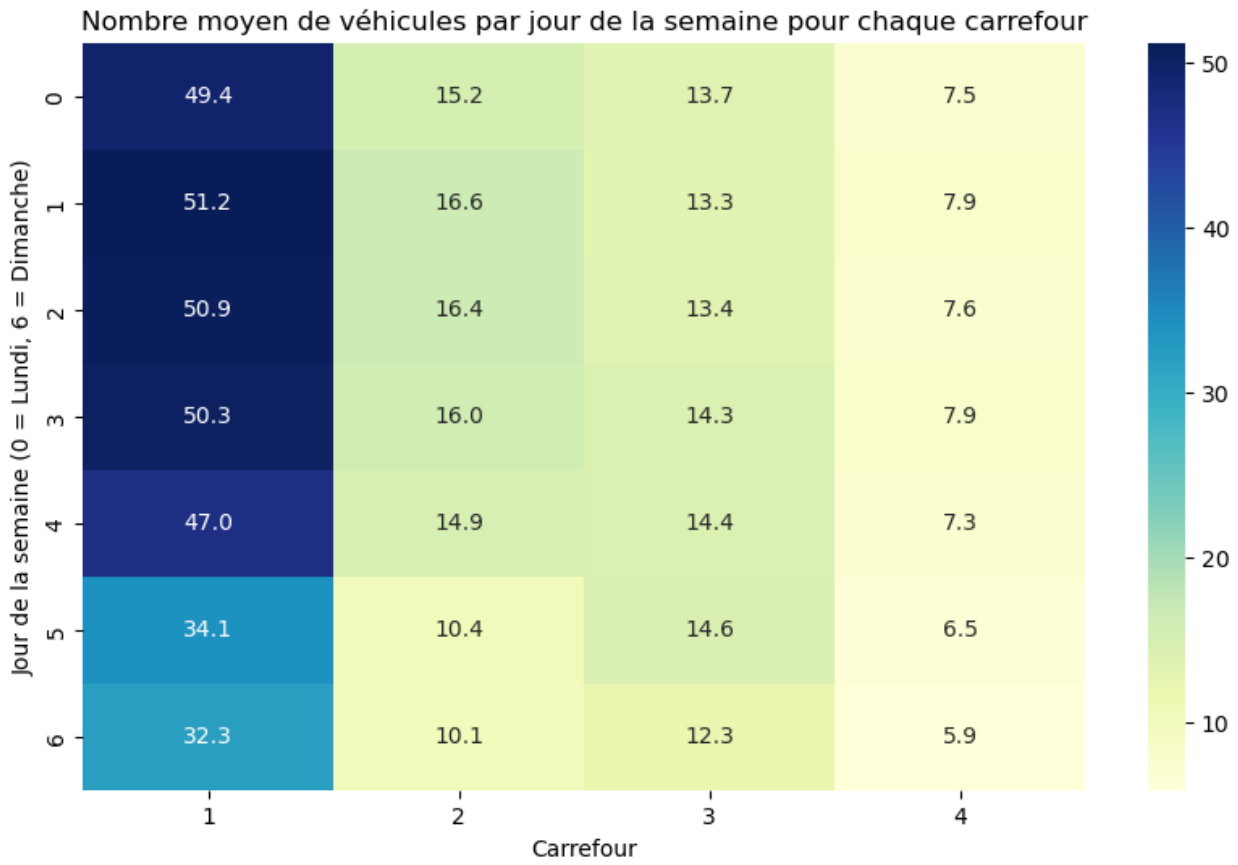
```
# Variation par carrefour
junction_traffic = data.groupby('Junction')['Vehicles'].mean()

plt.figure(figsize=(8, 5))
junction_traffic.plot(kind='bar', color='teal')
plt.title("Nombre moyen de véhicules par carrefour")
plt.xlabel("Carrefour")
plt.ylabel("Nombre moyen de véhicules")
plt.show()

# Variation saisonnière
data['Mois'] = data.index.month
monthly_traffic = data.groupby('Mois')['Vehicles'].mean()

plt.figure(figsize=(10, 6))
sns.lineplot(x=monthly_traffic.index, y=monthly_traffic.values,
marker="o", color="purple")
plt.title("Moyenne du nombre de véhicules par mois")
plt.xlabel("Mois")
```

```
plt.ylabel("Nombre moyen de véhicules")
plt.grid()
plt.show()
```


Nombre moyen de véhicules par carrefour
Nombre moyen de véhicules
0
10
20
30
40
1
2
3
4
Carrefour


Moyenne du nombre de véhicules par mois
Nombre moyen de véhicules
19
20
21
22
23
24
25
26
2
4
6
8
10
12
Mois

Résumé des tendances hebdomadaires:



Réponse3.Sources de variation :

Variations entre les carrefours, les périodes de la journée, et les jours fériés ou événements spéciaux.

Phase 2 : Prétraitement des Données

Bout de code:

```
scaler = MinMaxScaler()
data['Vehicles_normalized'] = scaler.fit_transform(data[['Vehicles']])
```

```
# Vérifier la normalisation
print("\nAperçu des données après normalisation :")
print(data[['Vehicles', 'Vehicles_normalized']].head())

# Sauvegarder les données prétraitées pour une utilisation ultérieure
data.to_csv("data_pretraitées.csv")
```

Explications des étapes :

1. **Nettoyage des Données** :
    a. Utilisation de l'interpolation temporelle (`method='time'`) pour remplir les valeurs manquantes en fonction des données disponibles avant et après les lacunes.
2. **Transformation des Données** :
    a. Extraction des caractéristiques temporelles :
        i. **Jour de la semaine** : Pour capturer les variations entre jours ouvrés et week-ends.
        ii. **Heure** : Pour analyser les variations horaires (pics matin/soir).
        iii. **Mois** et **Saison** : Pour identifier les tendances saisonnières.
    b. Ajout d'une colonne `Type_jour` pour distinguer les jours ouvrés des week-ends.
3. **Normalisation et Structuration** :
    a. Normalisation des valeurs de la colonne `Vehicles` pour faciliter l'apprentissage des modèles de séries temporelles. La normalisation est effectuée avec un `MinMaxScaler` pour ramener les données entre 0 et 1.
4. **Sauvegarde des Données Prétraitées** :
    a. Les données prétraitées sont sauvegardées dans un fichier CSV pour les étapes ultérieures du projet.

Résultats attendus :

- Les valeurs manquantes sont remplies.
- Les nouvelles caractéristiques temporelles (`Jour_semaine`, `Heure`, `Mois`, `Type_jour`, `Saison`) sont ajoutées.
- Les données sont normalisées pour un usage optimal dans des modèles de machine learning.

```
Valeurs manquantes avant traitement :
Junction    0
Vehicles    0
ID          0
dtype: int64

Valeurs manquantes après interpolation :
Junction    0
Vehicles    0
ID          0
dtype: int64

Aperçu des données avec les saisons précises :
                     Jour  Mois   Saison
DateTime
2015-11-01 00:00:00     1    11  Automne
2015-11-01 01:00:00     1    11  Automne
2015-11-01 02:00:00     1    11  Automne
2015-11-01 03:00:00     1    11  Automne
2015-11-01 04:00:00     1    11  Automne

Aperçu des données après normalisation :
                     Vehicles  Vehicles_normalized
DateTime
2015-11-01 00:00:00        15             0.078212
2015-11-01 01:00:00        13             0.067039
2015-11-01 02:00:00        10             0.050279
2015-11-01 03:00:00         7             0.033520
2015-11-01 04:00:00         9             0.044693
```

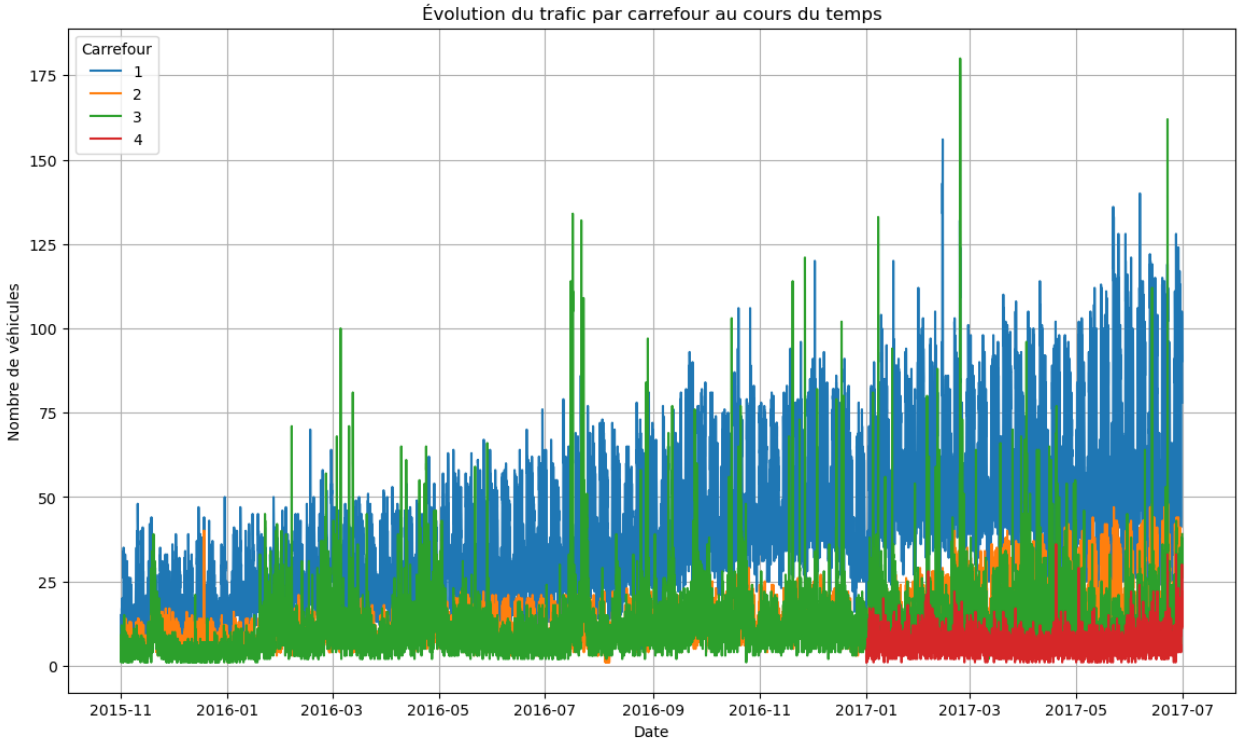
Phase 3 : AED

1. Visualisation des tendances

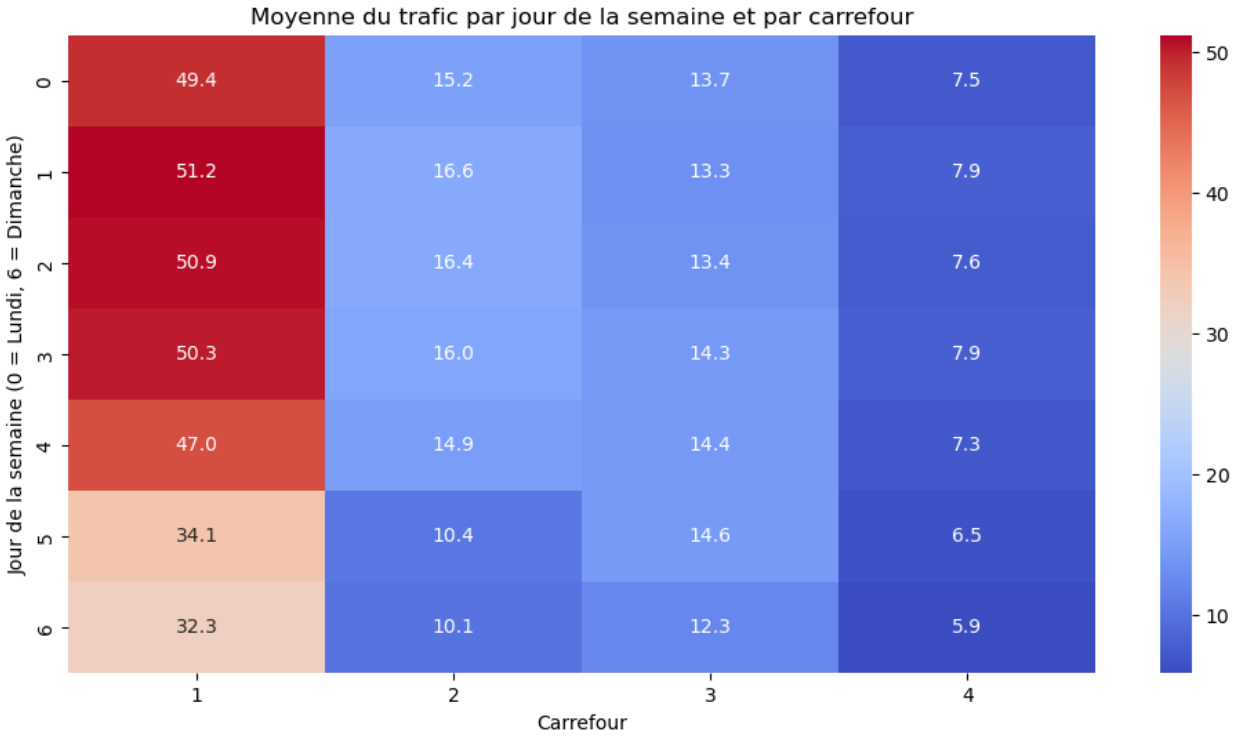
a. *Étapes :*

2. **Évolution du trafic par carrefour** :
    a. Un graphique linéaire est généré pour chaque carrefour (`Junction`).
    b. Les données sont groupées par carrefour pour montrer l'évolution temporelle du trafic.
3. **Trafic moyen par jour de la semaine** :
    a. Les données sont groupées par jour de la semaine (`Jour_semaine`) et carrefour.
    b. Un graphique en carte thermique (heatmap) montre les moyennes.
4. **Trafic moyen par heure de la journée** :
    a. Les données sont groupées par heure (`Heure`) et carrefour.
    b. Un graphique linéaire montre les variations horaires pour chaque carrefour.

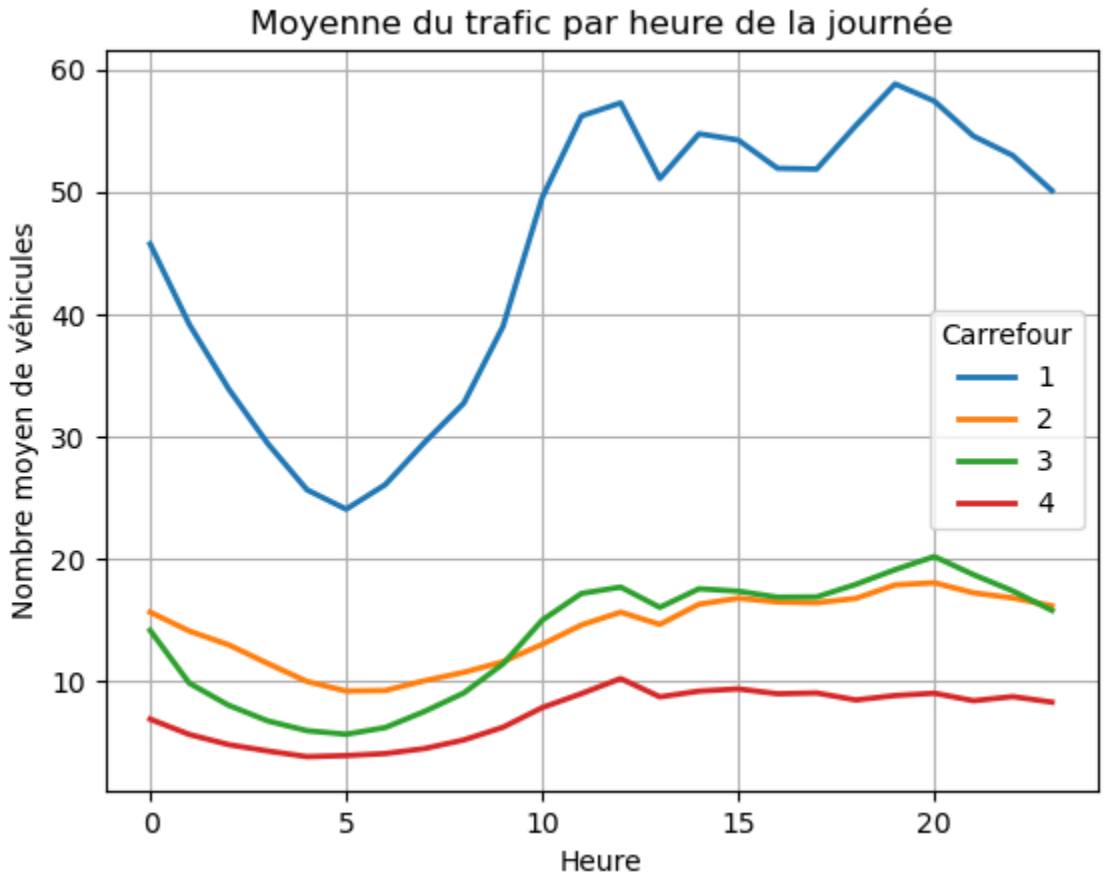
c. *Résultats attendus :*



5. **Graphique 1** : L'évolution temporelle du trafic pour chaque carrefour.



6. **Graphique 2** : Une carte thermique identifiant les jours de forte affluence.



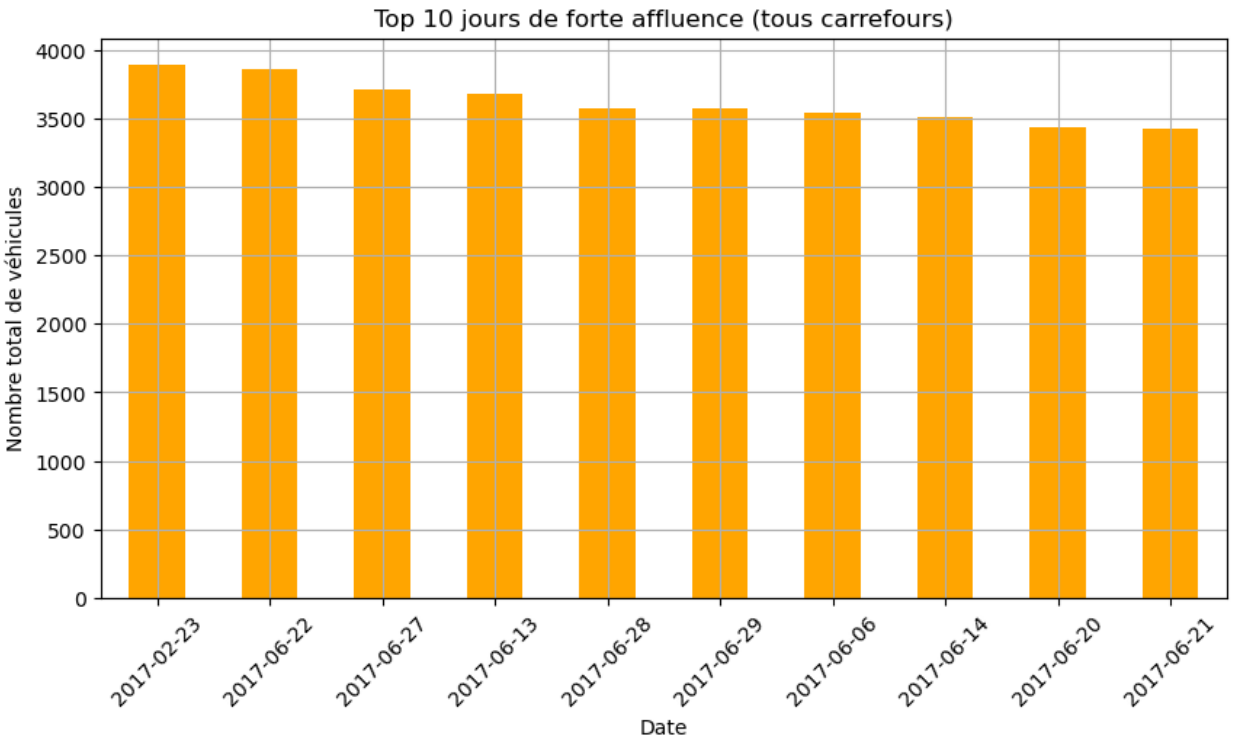
7. **Graphique 3** : Des pics horaires, par exemple, le matin et le soir.

2. Identification des périodes clés

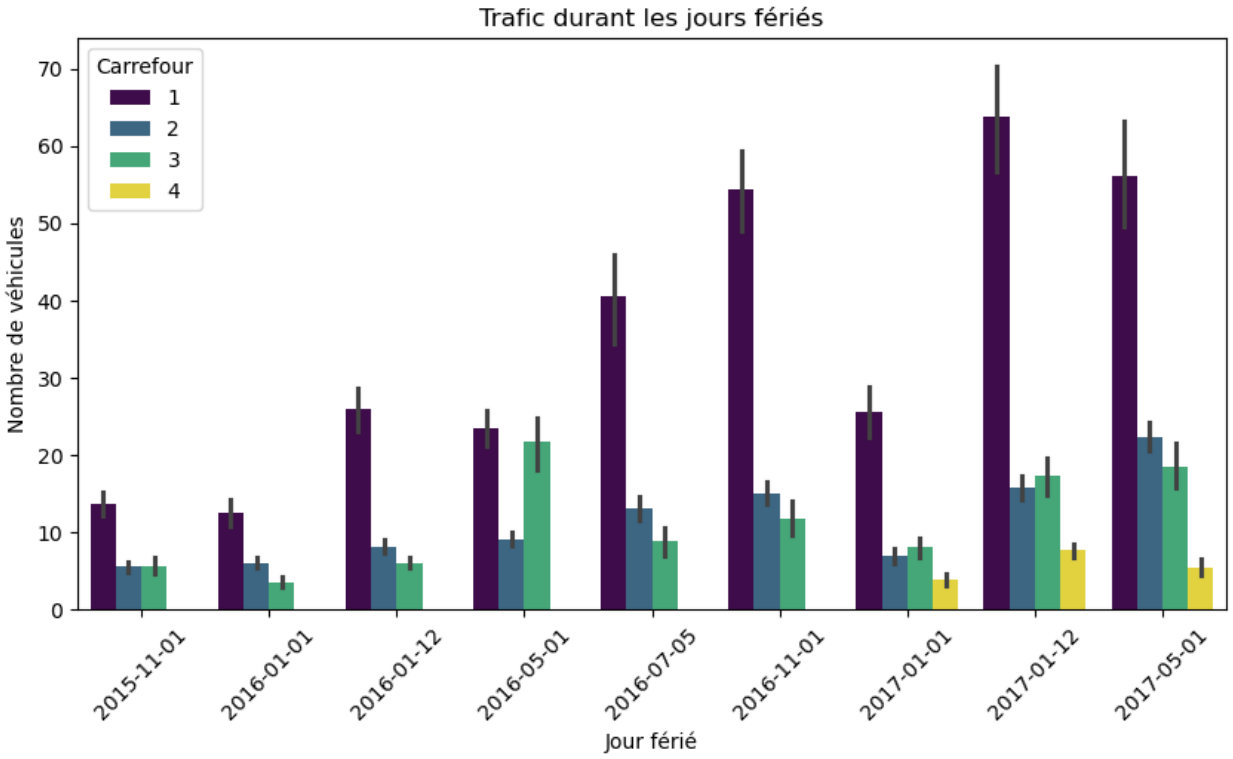
a. Étapes :

8. **Périodes de forte affluence (top 10 jours)** :
    a. Le trafic total est calculé pour chaque jour.
    b. Les 10 jours avec le trafic le plus élevé sont affichés dans un graphique en barres.
9. **Trafic durant les jours fériés** :
    a. Une colonne `Mois-Jour` est ajoutée pour extraire le mois et le jour des données.
    b. Les lignes correspondant aux jours fériés spécifiés sont filtrées.
    c. Un graphique en barres montre le trafic pour ces jours spécifiques.

d. Résultats attendus :



10. **Graphique 4** : Les 10 jours de trafic maximal, identifiant des événements ou pics inhabituels.



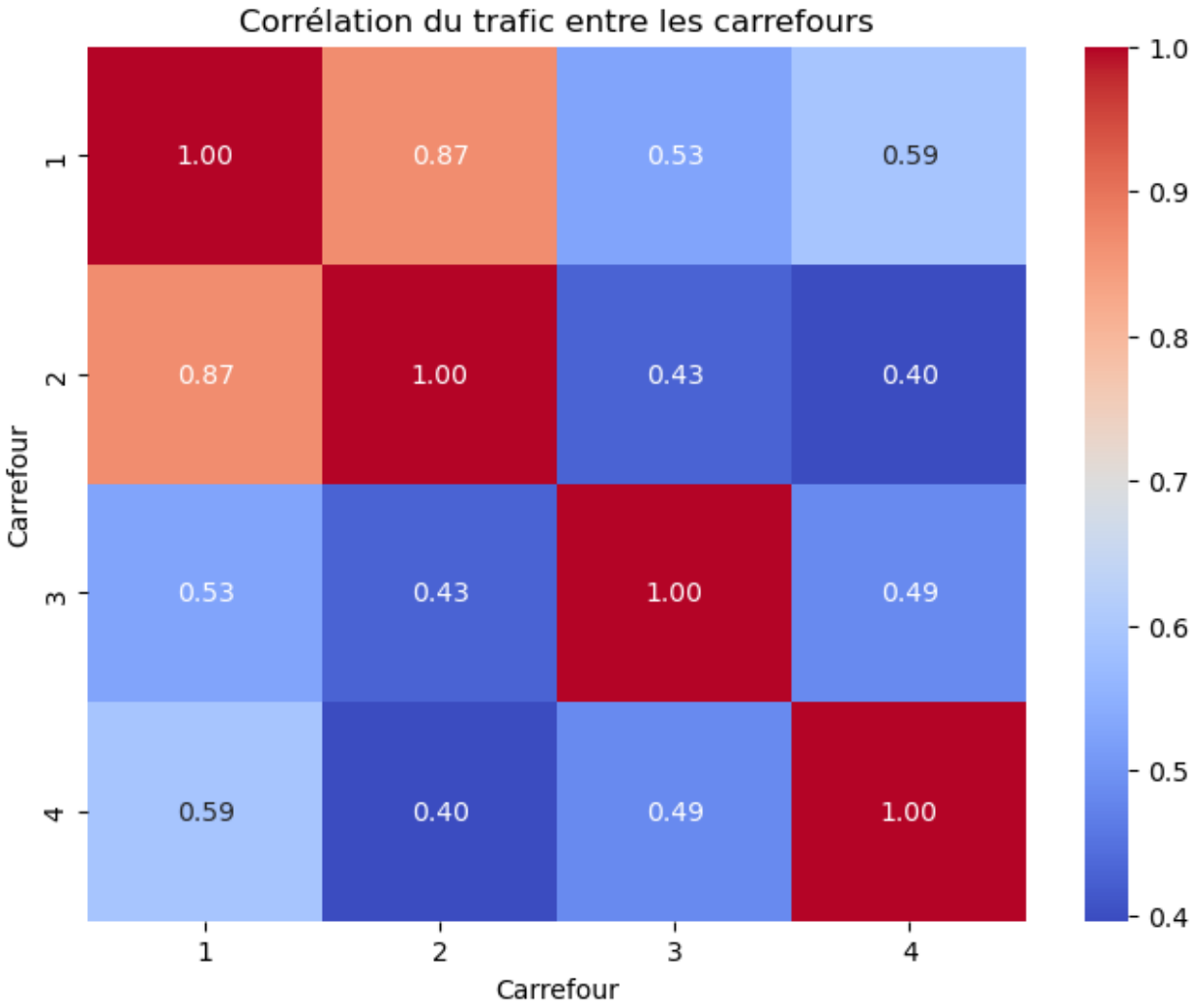
11. **Graphique 5** : Le trafic pour les jours fériés, par carrefour.

3. Corrélations et répartition

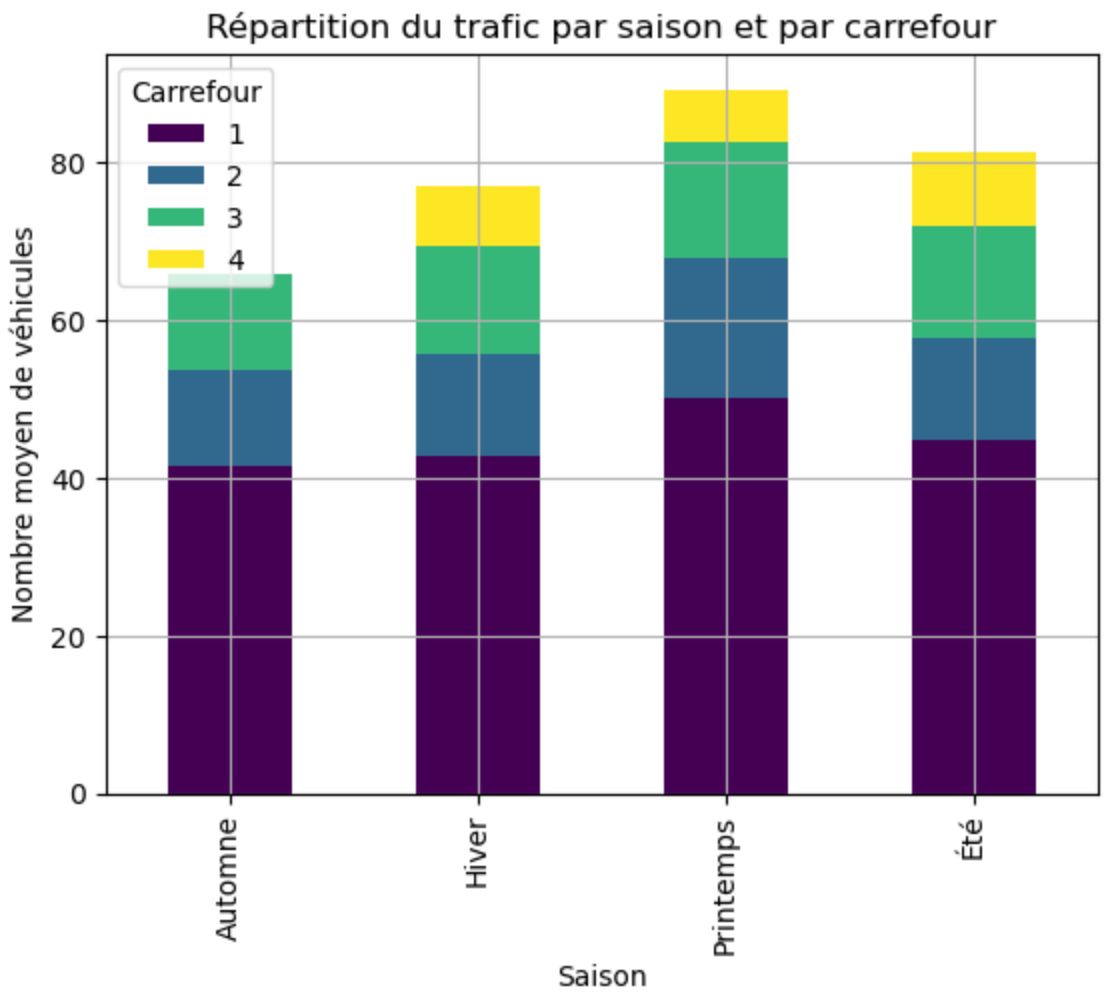
a. Étapes :

12. **Corrélation du trafic entre les carrefours** :
    a. Une matrice de corrélation est calculée pour le trafic entre les différents carrefours.
    b. Une carte thermique affiche les corrélations.
13. **Répartition du trafic par saison** :
    a. Les données sont groupées par saison et carrefour.
    b. Un graphique empilé montre la répartition du trafic par saison.

c. *Résultats attendus :*



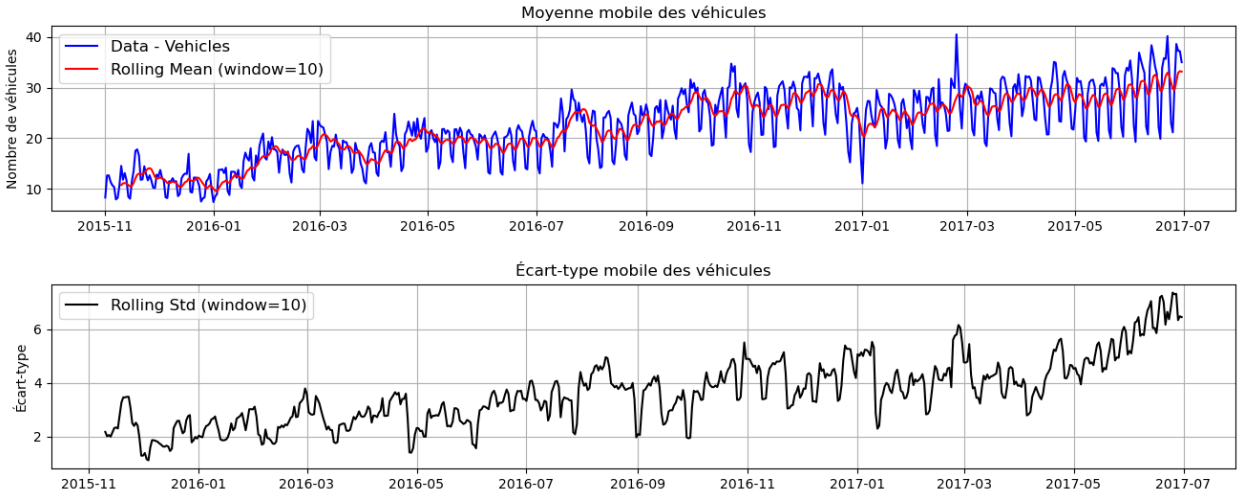
14. **Graphique 6** : Une carte thermique montrant des corrélations positives ou négatives entre les carrefours.



15. **Graphique 7** : La répartition saisonnière du trafic, permettant de repérer les variations dues aux conditions climatiques ou autres facteurs.

Phase 4 : Modélisation et Prédiction

ARRIMA



Test de stationnarité avec Dickey-Fuller

```
from statsmodels.tsa.stattools import adfuller
import pandas as pd

# 2. Test de stationnarité Dickey-Fuller
resultDFtest = adfuller(data_daily, autolag='AIC')

# 3. Résultats du test sous forme de série Pandas
Out = pd.Series(resultDFtest[0:4], index=['Test Statistic', 'p-value',
'#Lags Used', 'Number of Observations Used'])

# Ajouter les valeurs critiques (1%, 5%, 10%)
for key, value in resultDFtest[4].items():
    Out['Critical Value (%s)' % key] = value

# 4. Affichage des résultats
print('DICK-FULLER RESULTS: \n\n{}'.format(Out))

# 5. Interprétation
if resultDFtest[1] <= 0.05:
    print("\nConclusion : La série est stationnaire (p-value <=
0.05).")
else:
```

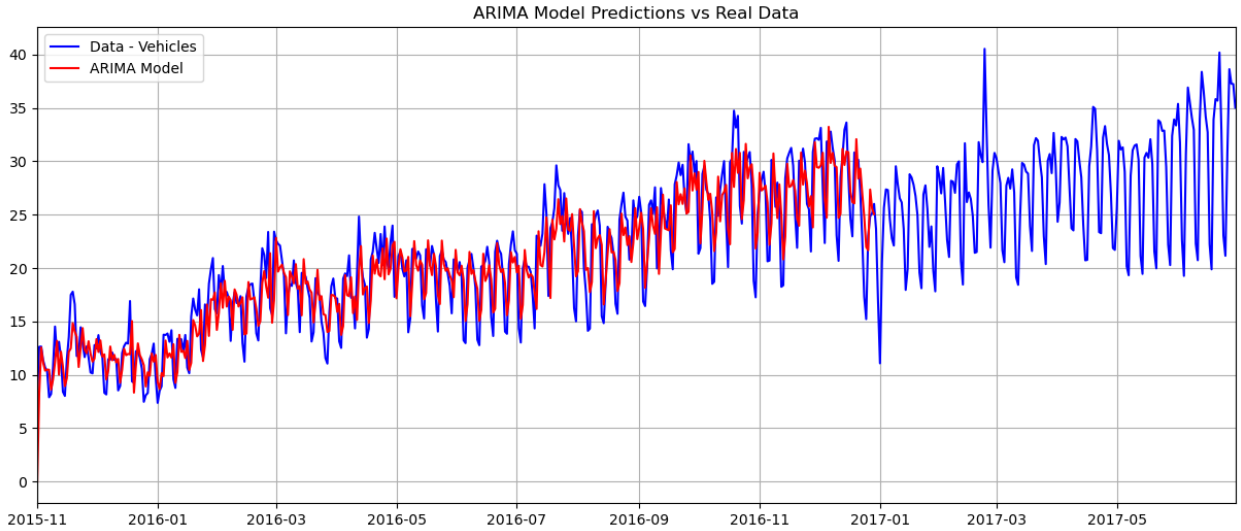
```
    print("\nConclusion : La série n'est pas stationnaire (p-value > 0.05).")
```

```
DICK-FULLER RESULTS:

Test Statistic                  -1.019845
p-value                          0.745957
#Lags Used                      19.000000
Number of Observations Used    588.000000
Critical Value (1%)             -3.441520
Critical Value (5%)             -2.866468
Critical Value (10%)            -2.569394
dtype: float64

Conclusion : La série n'est pas stationnaire (p-value > 0.05).
```

Training

```
# 1. Diviser la série temporelle en ensembles d'entraînement (70%) et de test (30%)
size = int(len(data_daily) * 0.7)  # 70% pour l'entraînement
train = data_daily[:size]  # Ensemble d'entraînement
test = data_daily[size:]  # Ensemble de test

# 2. Afficher les tailles des ensembles
print('Number of points in series:', len(data_daily))
print('Number of points in train:', len(train))
print('Number of points in test:', len(test))
```

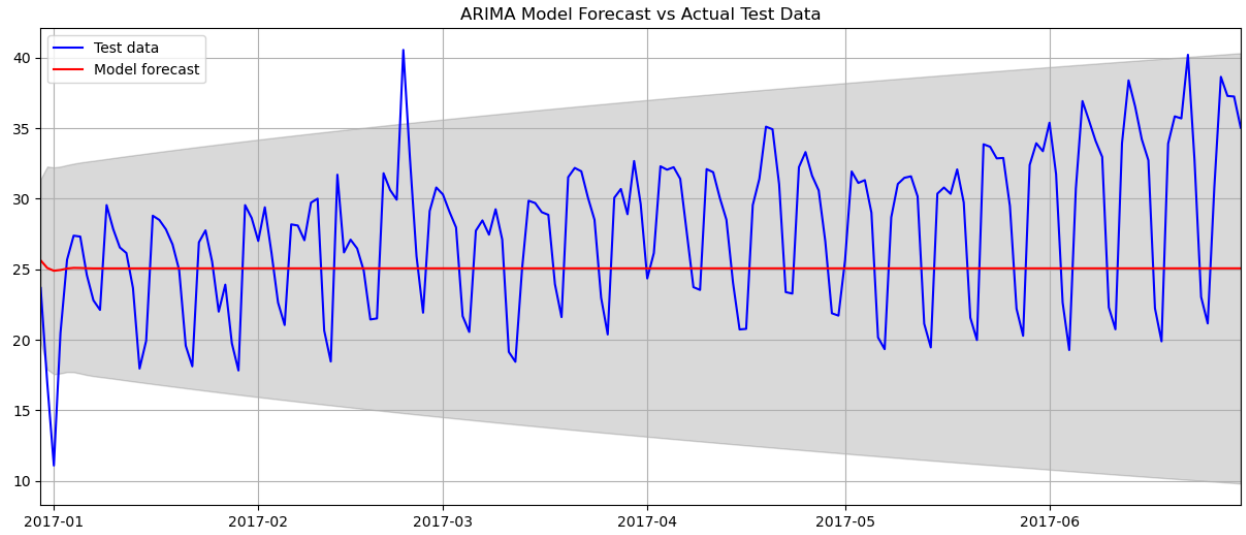
```
Number of points in series: 608
Number of points in train: 425
Number of points in test: 183
```



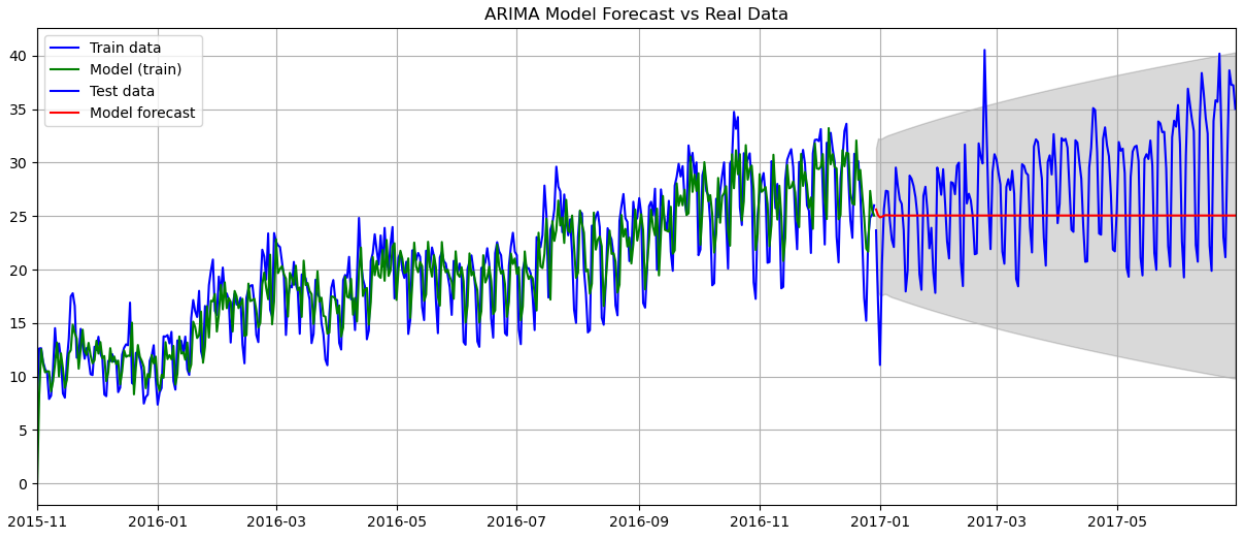

```
print(model_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
========
Dep. Variable:                Vehicles   No. Observations:
425
Model:                 ARIMA(2, 1, 1)   Log Likelihood                -
1060.170
Date:                Wed, 18 Dec 2024   AIC
2128.340
Time:                        00:18:12   BIC
2144.539
Sample:                    11-01-2015   HQIC
2134.740
                         - 12-29-2016
Covariance Type:                  opg
==============================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
--------
ar.L1          0.6178      0.051     12.049      0.000       0.517
0.718
ar.L2         -0.3375      0.064     -5.315      0.000      -0.462
-0.213
```

```
ma.L1          -0.8751      0.031    -27.847      0.000      -0.937
-0.814
sigma2          8.6705      0.609     14.247      0.000       7.478
9.863
===================================================================================
=============
Ljung-Box (L1) (Q):                   0.27   Jarque-Bera (JB):
1.50
Prob(Q):                              0.61   Prob(JB):
0.47
Heteroskedasticity (H):               2.19   Skew:
-0.15
Prob(H) (two-sided):                  0.00   Kurtosis:
3.01
===================================================================================
=============
```



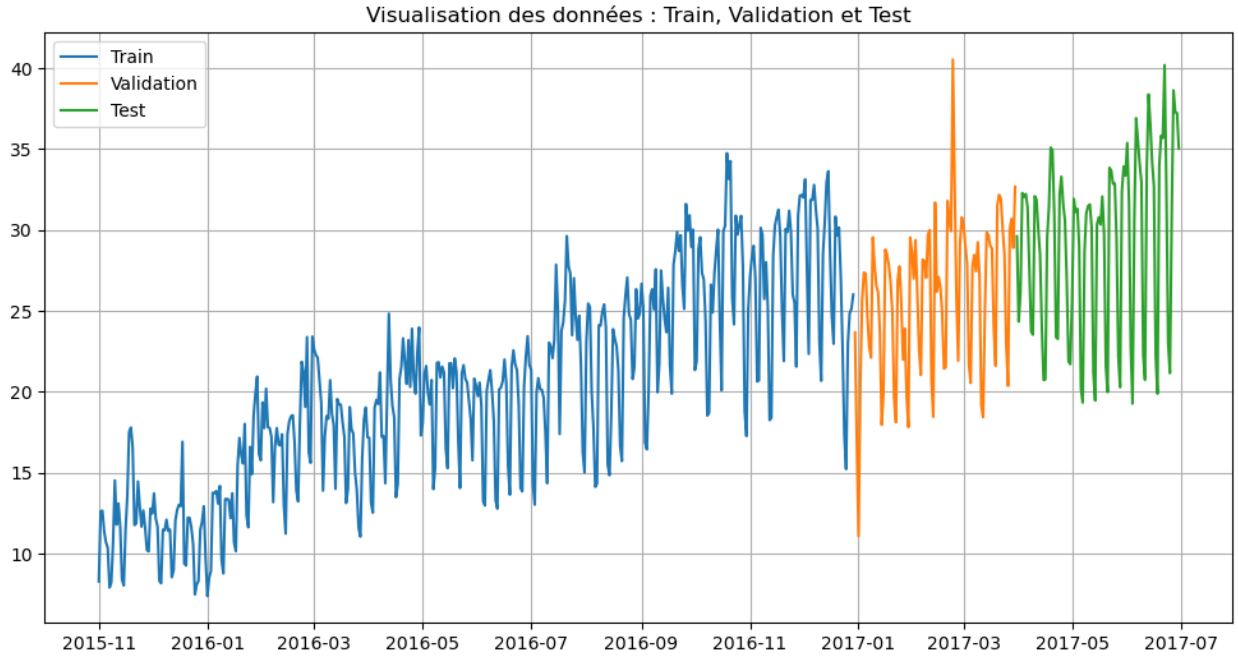
```
# 1. Visualisation des résultats
plt.figure(figsize=(15, 6))

# Tracer les données d'entraînement (en bleu)
plt.plot(train, c='blue', label='Train data')

# Tracer les prédictions sur l'ensemble d'entraînement (en vert)
plt.plot(model_fit.predict(dynamic=False), c='green', label='Model
(train)')

# Tracer les données de test (en bleu)
plt.plot(test, c='blue', label='Test data')

# Tracer les prévisions du modèle ARIMA pour l'ensemble de test (en
rouge)
plt.plot(forecast, c='red', label='Model forecast')

# Ajouter les intervalles de confiance autour des prévisions (bande
noire)
plt.fill_between(confidence.index, confidence['lower Vehicles'],
confidence['upper Vehicles'], color='k', alpha=0.15)

# Ajouter la légende, les axes et le titre
plt.legend()
plt.grid(True)
plt.margins(x=0)
plt.title("ARIMA Model Forecast vs Real Data")

# Afficher le graphique
plt.show()
```



LSTM

```
# # 1. Chargement des données et agrégation par jour
# data = pd.read_csv("data_pretraitées.csv", parse_dates=['DateTime'],
index_col='DateTime')
# data_daily = data['Vehicles'].resample('D').mean().dropna()

# 2. Fonction de préparation des séquences
def sampling(sequence, n_steps):
    X, Y = list(), list()
    for i in range(len(sequence)):
        sam = i + n_steps
        if sam > len(sequence) - 1:
            break
        x, y = sequence[i:sam], sequence[sam]
        X.append(x)
        Y.append(y)
    return np.array(X), np.array(Y)

# 3. Fonctions de normalisation
def MinMaxScale(t, t_or):
    return (t - t_or.min()) / (t_or.max() - t_or.min())

def InverseMinMaxScale(t, t_or):
    return t * (t_or.max() - t_or.min()) + t_or.min()
```

```python
# 4. Préparation des séquences avec 'sampling'
n_steps = 10  # Nombre d'étapes pour prédire
X, Y = sampling(data_daily.values, n_steps)

# 5. Division des données en Train, Validation et Test
size = int(len(data_daily) * 0.7)
size2 = int(((len(data_daily) - size) / 2) + size)

X_train, Y_train = X[:size], Y[:size]
X_val, Y_val = X[size:size2], Y[size:size2]
X_test, Y_test = X[size2:], Y[size2:]

# 6. Redimensionnement des données pour LSTM
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
X_val = X_val.reshape((X_val.shape[0], X_val.shape[1], 1))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))

# 7. Visualisation des ensembles de données
plt.figure(figsize=(12, 6))
plt.plot(data_daily[:size], label='Train')
plt.plot(data_daily[size:size2], label='Validation')
plt.plot(data_daily[size2:], label='Test')
plt.title("Visualisation des données : Train, Validation et Test")
plt.legend()
plt.grid()
plt.show()
```




```
Forme de X_train : (425, 10, 1)
Forme de Y_train : (425,)
Forme de X_val   : (91, 10, 1)
Forme de Y_val   : (91,)
```

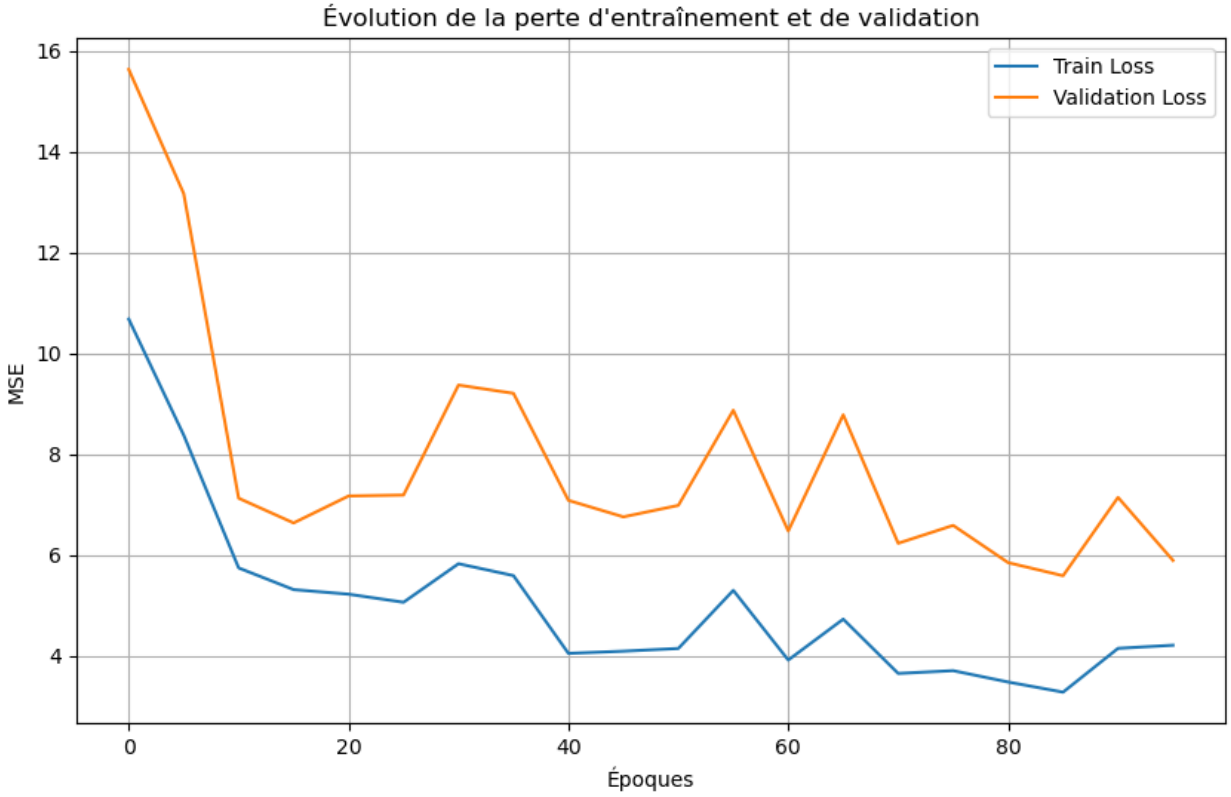
```
c:\Users\sayli\miniconda3\Lib\site-
packages\keras\src\layers\rnn\rnn.py:204: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_64"

Layer (type)	Output Shape	Param #
lstm_64 (LSTM)	(None, 50)	10,400

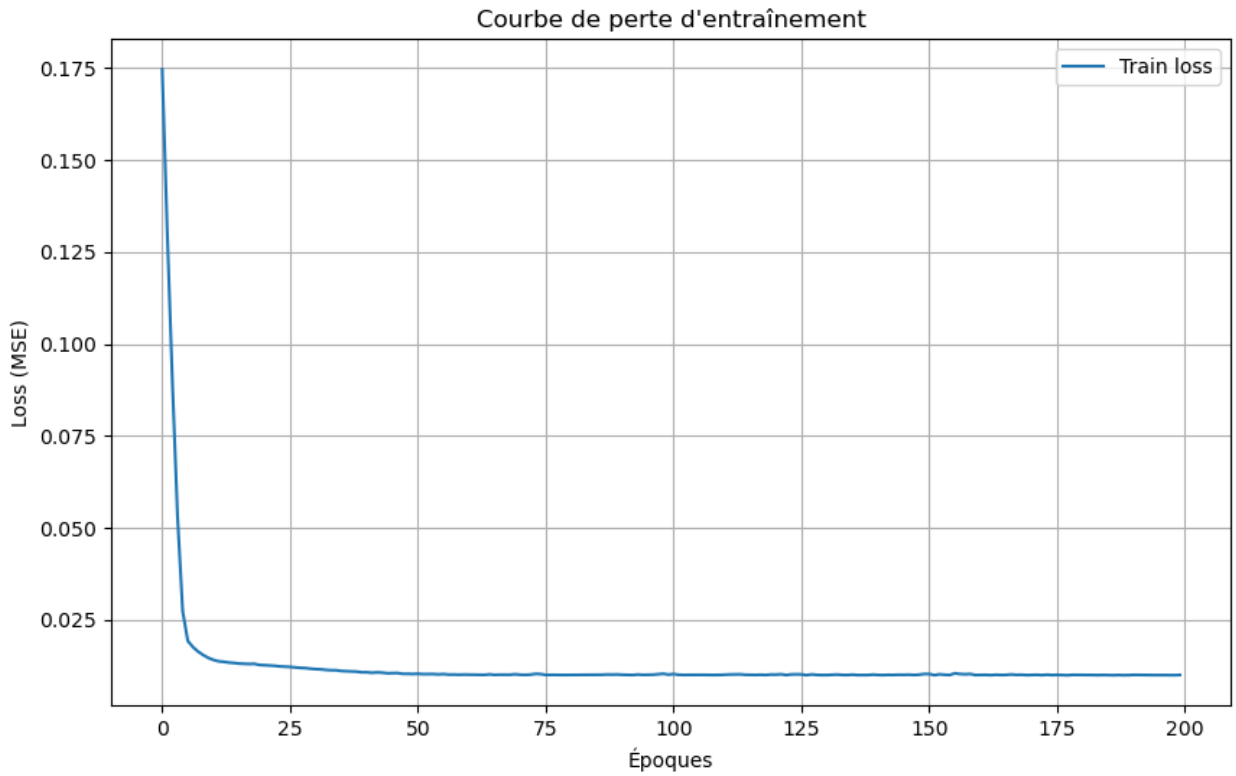
```
──────┤
│ dense_64 (Dense)                 │ (None, 1)              │
51 │
└──────────────────────────────────┴────────────────────────┴──────────
──────┘
```

Total params: 10,451 (40.82 KB)
Trainable params: 10,451 (40.82 KB)
Non-trainable params: 0 (0.00 B)




```python
# 7. Création du modèle LSTM
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, 1)))  # Couche LSTM
model.add(Dense(1))  # Couche de sortie
model.compile(optimizer='adam', loss='mse')  # Compilation du modèle

# 8. Entraînement du modèle
model_fit = model.fit(X_train, Y_train, epochs=200, verbose=1)
```

```python
# 9. Prédictions sur les ensembles d'entraînement et de test
Train_pred = model.predict(X_train, verbose=0)
Y_pred = model.predict(X_test, verbose=0)

# 10. Calcul des erreurs quadratiques moyennes (MSE)
print('MSE Train:', mean_squared_error(Train_pred, Y_train))  # MSE
sur l'entraînement
print('MSE Test:', mean_squared_error(Y_pred, Y_test))  # MSE sur le
test

# 11. Tracer la courbe de perte d'entraînement
plt.figure(figsize=(10, 6))
plt.plot(model_fit.history['loss'], label='Train loss')
plt.title("Courbe de perte d'entraînement")
plt.xlabel("Époques")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid()
plt.show()

# 12. Afficher la valeur minimale de la perte d'entraînement
print('Train MSE minimum:', min(model_fit.history['loss']))
```

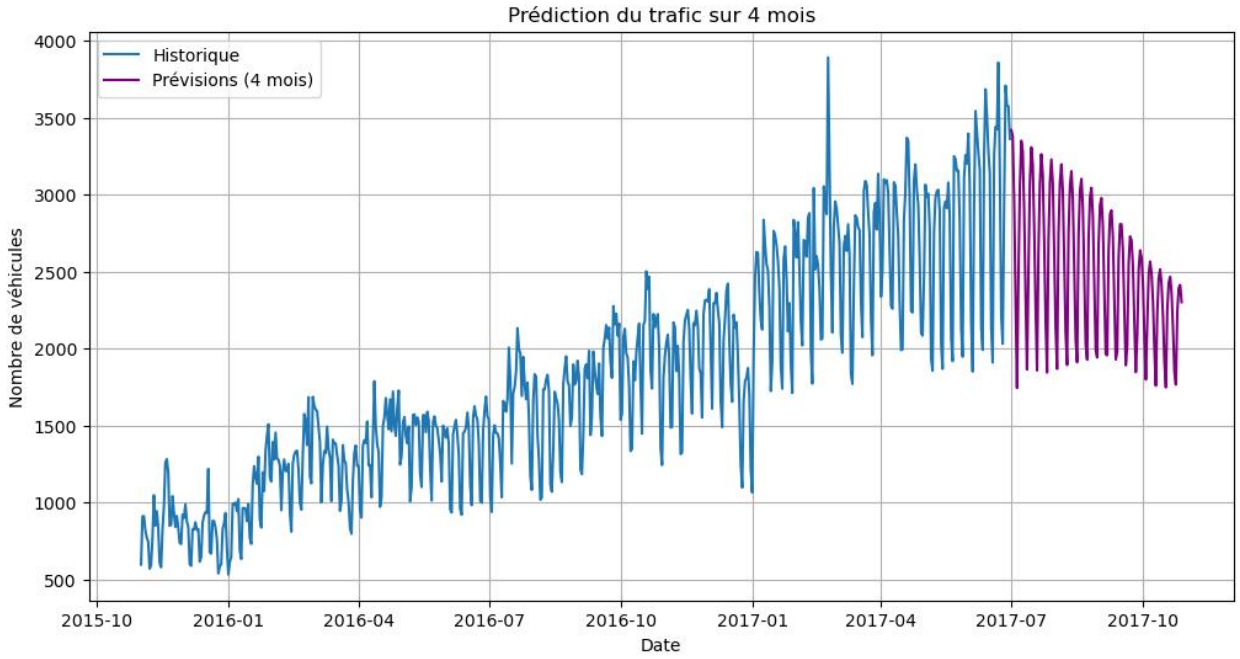


Prédiction:

```
# Génération des prévisions sur 4 mois (120 jours)
future_steps = 120
future_input = train_scaled[-seq_length:]  # Dernières séquences pour
prédire l'avenir

predictions = []
for _ in range(future_steps):
    pred = model.predict(future_input.reshape(1, seq_length, 1))
    predictions.append(pred[0, 0])
    future_input = np.append(future_input[1:],
pred).reshape(seq_length, 1)

# Rescaling des prédictions
predictions_rescaled =
scaler.inverse_transform(np.array(predictions).reshape(-1, 1))

# Affichage des prévisions
future_dates = pd.date_range(start=test.index[-1],
periods=future_steps + 1, freq='D')[1:]
```

```
plt.figure(figsize=(12, 6))
plt.plot(daily_data, label="Historique")
plt.plot(future_dates, predictions_rescaled, label="Prévisions (4
mois)", color='purple')
plt.title("Prédiction du trafic sur 4 mois")
plt.xlabel("Date")
plt.ylabel("Nombre de véhicules")
plt.legend()
plt.grid()
plt.show()
```

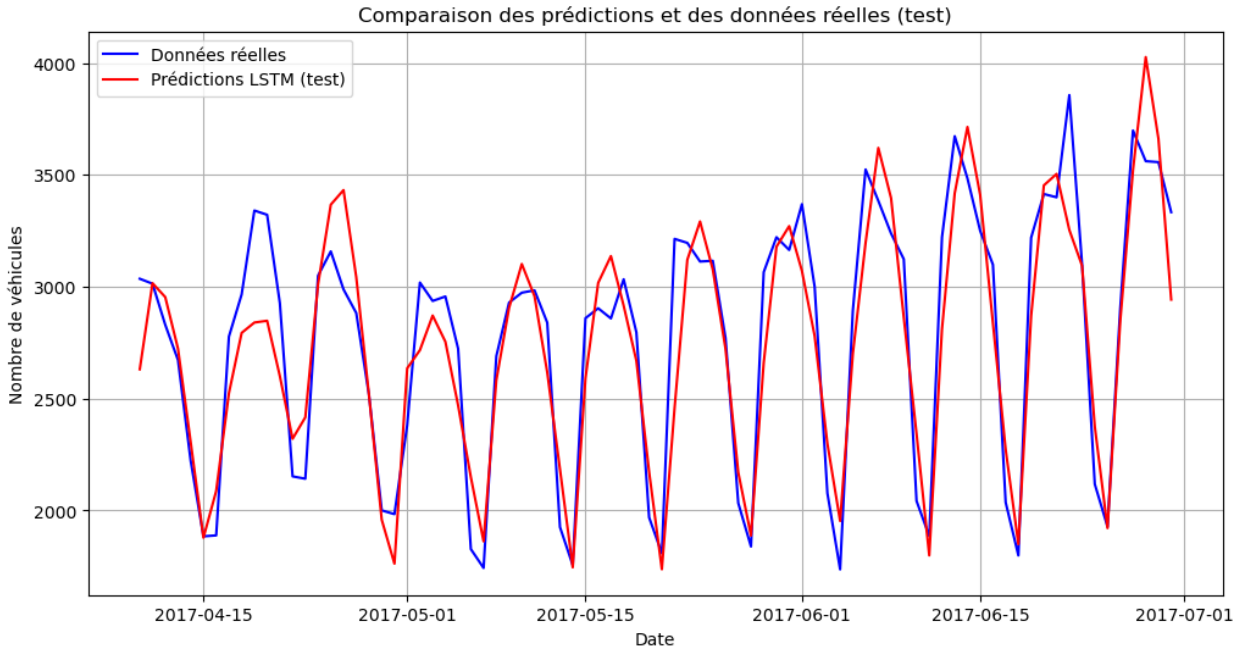
Résultats:



Phase 5 : Interprétation des Résultats et Recommandations

Précision et Fiabilité des Prédictions

Évaluation des performances du modèle sur les données de test :



Identification des Périodes Critiques

Top 10 jours avec le trafic le plus élevé prévu (4 mois) :

2017-08-12 00:00:00: 3152.85 véhicules

2017-08-05 00:00:00: 3197.19 véhicules

2017-07-16 00:00:00: 3197.85 véhicules

2017-07-29 00:00:00: 3229.02 véhicules

2017-07-22 00:00:00: 3263.01 véhicules

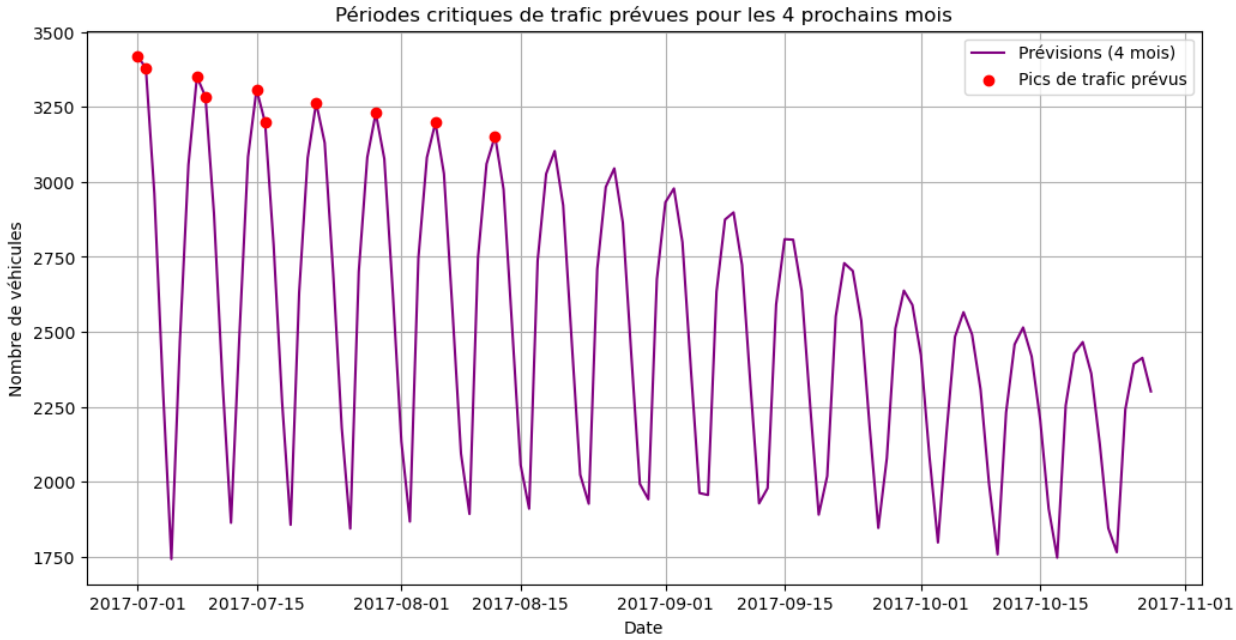
2017-07-09 00:00:00: 3283.90 véhicules

2017-07-15 00:00:00: 3308.07 véhicules

2017-07-08 00:00:00: 3350.48 véhicules

2017-07-02 00:00:00: 3379.88 véhicules

2017-07-01 00:00:00: 3419.67 véhicules


Périodes critiques de trafic prévues pour les 4 prochains mois
Prévisions (4 mois)
Pics de trafic prévus
Nombre de véhicules
Date
3500
3250
3000
2750
2500
2250
2000
1750
2017-07-01
2017-07-15
2017-08-01
2017-08-15
2017-09-01
2017-09-15
2017-10-01
2017-10-15
2017-11-01