Analyse de données et application d'algorithme de ML sur Employee Salaries en utilisant Spark

Matière : Big Data Analytics

Année Universitaires : 2023/2024

Réalisé par : **Hanae TAFZA**

Hiba ELAASSILI

Assia JALIDY

Encadrée par :

Mme. **Btihal EL GHALI**

Table des matières

INTRODUCTION

L'analyse de données et l'application d'algorithmes de machine Learning sur les salaires des employés sont des domaines cruciaux dans le paysage professionnel moderne.

Dans un monde où les entreprises traitent des volumes massifs de données, Spark se positionne comme une plateforme puissante pour la gestion et l'analyse de données distribuées. L'utilisation de techniques de machine learning (ML) devient de plus en plus incontournable pour comprendre les tendances, optimiser les coûts et prendre des décisions éclairées en matière de gestion des ressources humaines.

Dans cette perspective, l'objectif est d'explorer la manière dont Spark, un système de traitement de données distribuées, peut être employé pour analyser les données relatives aux salaires des employés. L'énorme quantité de données souvent associée à ces informations nécessite des outils efficaces et scalables, et c'est là que Spark se distingue. En utilisant Spark, nous pouvons traiter des ensembles de données massifs de manière parallèle, accélérant ainsi le processus d'analyse.

En intégrant des algorithmes de machine learning dans cette démarche, nous chercherons à découvrir des modèles cachés, à prédire des tendances futures et à fournir des recommandations basées sur des analyses prédictives. L'application de modèles de ML sur les salaires des employés peut permettre aux organisations d'optimiser leur structure de rémunération, d'identifier des anomalies, et même de prédire les évolutions futures en fonction de divers paramètres.

Ainsi, cette exploration de l'analyse de données et de l'application d'algorithmes de machine learning sur les salaires des employés à l'aide de Spark représente une démarche moderne et technologiquement avancée pour tirer des informations significatives, favorisant ainsi une prise de décision plus éclairée et stratégique dans le domaine des ressources humaines.

I. Contexte du Projet :

1. Descriptif du projet :

Le projet s'inscrit dans le domaine de l'analyse de données et de l'application d'algorithmes de machine Learning dans le contexte spécifique des salaires des employés. À mesure que les organisations traitent des volumes croissants de données liées aux ressources humaines, il devient impératif d'adopter des approches analytiques avancées pour extraire des informations significatives. Dans ce contexte, l'analyse des salaires par le biais de techniques de machine Learning offre une opportunité précieuse d'optimiser la gestion des ressources humaines, d'anticiper les tendances et de prendre des décisions éclairées.

2. Objectifs de l'Analyse de Données et du Machine Learning :

Les principaux objectifs de ce projet sont les suivants :

- **Compréhension des Tendances Salariales** : Analyser les données des salaires pour identifier les tendances, les variations et les modèles significatifs qui peuvent influencer les décisions en matière de rémunération.
- **Prédiction des Salaires** : Mettre en œuvre des algorithmes de machine Learning pour élaborer des modèles prédictifs capables d'estimer les salaires des employés en fonction de différentes variables.
- **Optimisation des Ressources Humaines** : Fournir des recommandations basées sur l'analyse des salaires pour optimiser la structure de rémunération, équilibrer les coûts et maximiser la satisfaction des employés.

3. Justification de l'utilisation de Spark :

Spark est choisi comme cadre de travail pour ce projet en raison de ses avantages clés dans le traitement de données massives et distribuées. Les données salariales souvent volumineuses exigent une plateforme capable de traiter efficacement des ensembles de données, de plus, Spark propose des bibliothèques MLlib dédiées à la machine Learning. Ainsi, l'utilisation de PySpark pour effectuer des prédictions sur l'ensemble de données des salaires des employés offre une approche technologiquement avancée et efficace pour atteindre les objectifs du projet.

II. Analyse Exploratoire des Données :

1. Description de la dataset :

Il s'agit d'un ensemble de données sur les salaires des employés. Il comprend 3 fichiers CSV, dont 2 sont destinés à l'entraînement. Cet ensemble de données contient 1 000 000 d'échantillons et fournie des informations sur les employés, en particulier aux détails des emplois, à l'éducation, à l'expérience professionnelle, à l'emplacement et aux salaires. Voici un descriptif des colonnes :

- **Job Id** : un identifiant unique associé à chaque emploi.
- **Company Id** : variable catégorielle, un identifiant de l'entreprise à laquelle l'emploi est associé.
- **Job Type** : variable catégorielle, le type d'emploi occupé par l'individu (ex. CFO, CEO, VICE_PRESIDENT, MANAGER, etc.).
- **Degree** : variable catégorie, le niveau de diplôme détenu par l'individu (ex. HIGH_SCHOOL, BACHELORS, MASTERS, DOCTORAL, etc.).
- **Major** : variable catégorielle, le domaine d'études principal de l'individu (ex. MATH, NONE, PHYSICS, CHEMISTRY, etc.).
- **Industry** : variable catégorielle, le secteur industriel auquel l'emploi est associé (ex. HEALTH, WEB, AUTO, FINANCE, etc.).
- **Years Experience** : variable numérique (entier), le nombre d'années d'expérience professionnelle de l'individu.
- **Miles From Metropolis** : variable numérique (entier), la distance en miles de l'emplacement de l'individu jusqu'au centre métropolitain le plus proche.
- **Salary** : variable numérique (entier), le salaire associé à l'emploi.

2. Installation des librairies nécessaires et Création de l'environnement Spark :

Le code suivant utilise PySpark pour l'analyse de données, et matplotlib et seaborn pour la visualisation, voici une brève explication de chaque ligne :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
```

- **From pyspark.sql import SparkSession** : importe la classe SparkSession de la bibliothèque PySpark, qui est nécessaire pour travailler avec des données structurées en utilisant Spark.
- **From pyspark.sql.functions import col** : importe la fonction col de la bibliothèque PySpark qui est utilisée pour faire référence à des colonnes dans les opérations sur les données Spark.

- **Import matplotlib.pyplot as plt** : importe la bibliothèque Matplotlib, qui est une bibliothèque de visualisation en Python.
- **Import seaborn as sns** : importe la bibliothèque Seaborn, qui est une extension de Matplotlib pour la visualisation statistique.
- **Import pandas as pd** : importe la bibliothèque Pandas, qui est utilisée pour la manipulation et l'analyse de données en Python.

Le code suivant crée une session Spark à l'aide de la classe SparkSession pour établir une

```
# Create a Spark session
spark = SparkSession.builder.appName("JobDataEDA").getOrCreate()
```

connexion à l'environnement Spark voici une explication détaillée :

- **SparkSession.builder** : la méthode «builder» est utilisée pour configurer et créer une nouvelle session Spark.
- **AppName ("JobDataEDA")** : la méthode appName définit un nom convivial pour l'application Spark , dans notre cas, l'application est nommée "JobDataEDA".
- **GetOrCreate** : la méthode getOrCreate tente de récupérer une session Spark existante avec le nom spécifié "JobDataEDA". Si aucune session avec ce nom n'existe, elle crée une nouvelle session.
- **Spark** : la variable spark est assignée à la session Spark nouvellement créée ou récupérée.

3. Traitement des Données :

3.1 Chargement du jeu de données :

Le code suivant utilise la bibliothèque Pandas pour charger deux ensembles de données à partir de fichiers CSV, les fusionner en fonction d'une colonne commune.

```
import pandas as pd

# Load the datasets
data = pd.read_csv("/content/train_dataset.csv")
data2 = pd.read_csv("/content/train_salaries.csv")

# Specify the common column for the join
common_column = "jobId"

# Merge the datasets based on the common column
merged_data = pd.merge(data, data2, on=common_column)

# Specify the columns to keep
columns_to_keep = ["jobId", "companyId", "jobType", "degree", "major", "industry", "yearsExperience", "milesFromMetropolis", "salary"]

# Select the desired columns
result_df = merged_data[columns_to_keep]

# Display the result
print(result_df.head())
```

```
             jobId companyId         jobType       degree      major  \
0  JOB1362684407687    COMP37             CFO      MASTERS       MATH
1  JOB1362684407688    COMP19             CEO  HIGH_SCHOOL       NONE
2  JOB1362684407689    COMP52  VICE_PRESIDENT     DOCTORAL    PHYSICS
3  JOB1362684407690    COMP38         MANAGER     DOCTORAL  CHEMISTRY
4  JOB1362684407691     COMP7  VICE_PRESIDENT    BACHELORS    PHYSICS

  industry  yearsExperience  milesFromMetropolis  salary
0   HEALTH               10                   83     130
1      WEB                3                   73     101
2   HEALTH               10                   38     137
3     AUTO                8                   17     142
4  FINANCE                8                   16     163
```

Interprétation : Le résultat est sous forme d'un tableau présentant les données après fusion des deux ensembles de données en fonction de la colonne "jobId". Les colonnes incluent des informations telles que l'identifiant de l'emploi ("jobId"), le nom de l'entreprise ("companyId"), le type d'emploi ("jobType"), le niveau de diplôme ("degree"), la spécialisation ("major"), le secteur industriel ("industry"), l'expérience en années ("yearsExperience"), la distance jusqu'au centre métropolitain en miles ("milesFromMetropolis"), et le salaire ("salary").

3.2 Fusion du train_dataset avec train_salaries :

Avant de commencer l'analyse, nous allons joindre les deux dataframes sur la colonne "jobId". si nous joignons les deux dataframes tels qu'ils sont, nous aurons deux colonnes identiques "jobId", ce qui créera une ambiguïté. Pour cette raison, nous allons renommer la colonne "JobId" dans le deuxième dataframe, afin que nous puissions la supprimer après la jointure sans créer d'ambiguïté avec la colonne "jobId" du premier dataframe.

Le code suivant converti le dataFrame Pandas en un dataFrame PySpark pour tirer parti des fonctionnalités de traitement distribué de Spark. La dernière ligne est utilisée pour afficher quelques lignes du dataFrame PySpark dans la console.

```
from pyspark.sql import SparkSession

# Assuming 'spark' is your SparkSession
spark = SparkSession.builder.appName("example").getOrCreate()

# Convert pandas DataFrame to PySpark DataFrame
spark_df = spark.createDataFrame(result_df)

# Show the first few rows
spark_df.show()
```

```
+----------+---------+-------------+----------+----------+---------+--------------+------------------+------+
|     jobId|companyId|      jobType|    degree|     major| industry|yearsExperience|milesFromMetropolis|salary|
+----------+---------+-------------+----------+----------+---------+--------------+------------------+------+
|JOB1362684407687|   COMP37|          CFO|   MASTERS|      MATH|   HEALTH|            10|                83|   130|
|JOB1362684407688|   COMP19|          CEO|HIGH_SCHOOL|      NONE|      WEB|             3|                73|   101|
|JOB1362684407689|   COMP52|VICE_PRESIDENT|  DOCTORAL|   PHYSICS|   HEALTH|            10|                38|   137|
|JOB1362684407690|   COMP38|      MANAGER|  DOCTORAL| CHEMISTRY|     AUTO|             8|                17|   142|
|JOB1362684407691|    COMP7|VICE_PRESIDENT| BACHELORS|   PHYSICS|  FINANCE|             8|                16|   163|
|JOB1362684407692|   COMP15|      MANAGER|  DOCTORAL|   COMPSCI|  FINANCE|             2|                31|   113|
|JOB1362684407693|   COMP15|          CFO|      NONE|      NONE|   HEALTH|            23|                24|   178|
|JOB1362684407694|   COMP24|       JUNIOR| BACHELORS| CHEMISTRY|EDUCATION|             9|                70|    73|
|JOB1362684407695|   COMP20|      JANITOR|HIGH_SCHOOL|      NONE|EDUCATION|             1|                54|    31|
|JOB1362684407696|   COMP41|VICE_PRESIDENT| BACHELORS| CHEMISTRY|     AUTO|            17|                68|   104|
|JOB1362684407697|   COMP56|      JANITOR|HIGH_SCHOOL|      NONE|   HEALTH|            24|                30|   102|
|JOB1362684407698|    COMP7|          CEO|   MASTERS|   PHYSICS|EDUCATION|             7|                79|   144|
|JOB1362684407699|    COMP4|       JUNIOR|      NONE|      NONE|      OIL|             8|                29|    79|
|JOB1362684407700|   COMP54|       JUNIOR|   MASTERS|      MATH|  FINANCE|            21|                26|   193|
|JOB1362684407701|   COMP57|      JANITOR|      NONE|      NONE|     AUTO|            21|                81|    47|
|JOB1362684407702|   COMP20|          CTO|   MASTERS|   BIOLOGY|  SERVICE|            13|                 8|   172|
|JOB1362684407703|   COMP14|       JUNIOR|   MASTERS|   PHYSICS|  SERVICE|             1|                91|    47|
|JOB1362684407704|   COMP61|VICE_PRESIDENT|   MASTERS|LITERATURE|  SERVICE|            23|                43|   126|
|JOB1362684407705|   COMP58|          CEO|   MASTERS|LITERATURE|  SERVICE|            23|                66|   122|
|JOB1362684407706|    COMP3|          CEO|   MASTERS|   PHYSICS|EDUCATION|             9|                99|    95|
+----------+---------+-------------+----------+----------+---------+--------------+------------------+------+
only showing top 20 rows
```

Le code suivant montre que 'df' est la dataFrame PySpark, utilise en réalité le dataFrame Pandas result_df, cela permet de visualiser rapidement la taille du dataFrame en termes de lignes et de colonnes.

```
[9] # Assuming 'df' is your PySpark DataFrame
    num_rows = result_df.count()
    num_columns = len(result_df.columns)

    print("Number of rows:", num_rows)
    print("Number of columns:", num_columns)
```

```
Number of rows: jobId                  1000000
companyId              1000000
jobType                1000000
degree                 1000000
major                  1000000
industry               1000000
yearsExperience        1000000
milesFromMetropolis    1000000
salary                 1000000
dtype: int64
Number of columns: 9
```

Interprétation : Le résultat affiché indique que dataFrame Pandas (result_df) contient un total de 1.000.000 lignes et 9 colonnes, chaque ligne représente un enregistrement, et chaque colonne représente une variable associée à cet échantillon.

Le code suivant utilise PySpark pour vérifier la présence de valeurs manquantes dans le dataFrame result_df et affiche le nombre de valeurs manquantes pour chaque colonne.

Le résultat affichera une table avec les colonnes du DataFrame d'origine et les valeurs true ou false indiquant si chaque cellule est nulle ou non.

```
from pyspark.sql.functions import col

# Check for missing values in the entire DataFrame
missing_values = result_df.select([col(c).isNull().alias(c) for c in result_df.columns])

# Display the missing values count for each column
missing_values.show()
```

```
+-----+---------+-------+------+-----+--------+---------------+-------------------+------+
|jobId|companyId|jobType|degree|major|industry|yearsExperience|milesFromMetropolis|salary|
+-----+---------+-------+------+-----+--------+---------------+-------------------+------+
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
|false|    false|  false| false|false|   false|          false|              false| false|
+-----+---------+-------+------+-----+--------+---------------+-------------------+------+
only showing top 20 rows
```

Interprétation : Le résultat affiché indique qu'il n'y a pas de valeurs manquantes (valeurs nulles) dans le DataFrame result_df. Chaque cellule du DataFrame **missing_values** affiche false, ce qui signifie que toutes les valeurs dans le DataFrame result_df sont présentes et non nulles.

3.3 Calcul des Statistiques Descriptives :

Le code utilise PySpark pour calculer des statistiques sommaires sur les colonnes numériques spécifiées (yearsExperience, milesFromMetropolis, et salary) du DataFrame result_df.

```
# Summary statistics for numerical columns
result_df.describe(['yearsExperience', 'milesFromMetropolis','salary']).show()
```

```
+-------+-----------------+------------------+------------------+
|summary|  yearsExperience|milesFromMetropolis|            salary|
+-------+-----------------+------------------+------------------+
|  count|          1000000|           1000000|           1000000|
|   mean|        11.992386|          49.52926|        116.061818|
| stddev|7.212390868885695|28.877732628719937|38.717936381133605|
|    min|                0|                 0|                 0|
|    max|               24|                99|               301|
+-------+-----------------+------------------+------------------+
```

Interprétation : Le résultat affiché présente les statistiques sommaires pour les colonnes

- **count :** Chaque colonne compte 100 000 entrées, ce qui indique que le DataFrame contient 100 000 lignes de données.
- **mean** : La moyenne des années d'expérience est d'environ 11,99 ans, la distance moyenne depuis la métropole est d'environ 49,53 miles, et le salaire moyen est d'environ 116,062 dollars.
- **stddev** : La déviation standard (qui mesure la dispersion des données) est d'environ 7,21 pour les années d'expérience, d'environ 28,88 pour la distance depuis la métropole, et d'environ 38,72 pour le salaire.
- **min :** Les valeurs minimales pour les années d'expérience, la distance depuis la métropole et le salaire sont 0, 0 et 0 respectivement.
- **max :** Les valeurs maximales sont de 24 ans pour l'expérience, 99 miles pour la distance depuis la métropole et 301 dollars pour le salaire.

3.4 Afficher le schéma du DataFrame :

La fonction printSchema () en PySpark est utilisée pour afficher le schéma du DataFrame, montrant les types de données et la structure des colonnes.

```
[13] result_df.printSchema()
```

```
root
 |-- jobId: string (nullable = true)
 |-- companyId: string (nullable = true)
 |-- jobType: string (nullable = true)
 |-- degree: string (nullable = true)
 |-- major: string (nullable = true)
 |-- industry: string (nullable = true)
 |-- yearsExperience: long (nullable = true)
 |-- milesFromMetropolis: long (nullable = true)
 |-- salary: long (nullable = true)
```

Interprétation : Le schéma indique les noms des colonnes, leurs types de données, et si les valeurs nulles sont autorisées (**nullable = true**).

- **JobId, companyId, jobType, degree, major, industry** : string (chaîne de caractères)
- **YearsExperience, milesFromMetropolis, salary** : long (entier long)

En imprimant le schéma du dataframe, nous pouvons voir que certaines caractéristiques telles que le salaire (salary), l'expérience en années (yearsExperience) et la distance depuis la métropole (milesFromMetropolis) devraient être converties en type entier.

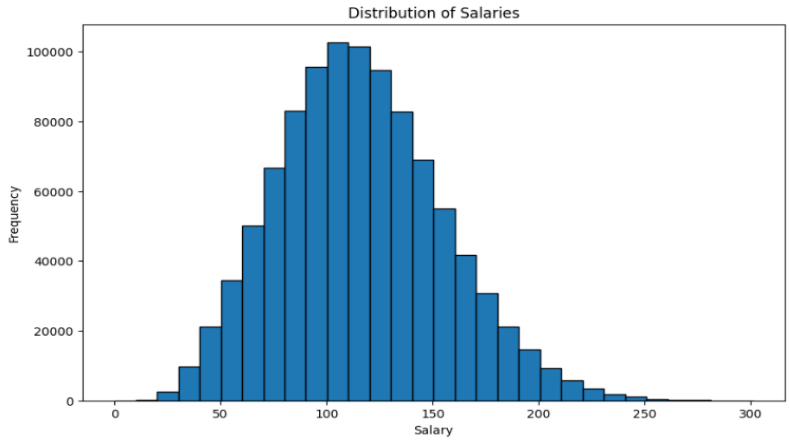
III. Visualisation des Résultat :

1. Distribution des salaires à l'aide d'un histogramme :

Ce code effectue une analyse descriptive des salaires, en fournissant des statistiques sommaires et en visualisant la distribution des salaires à l'aide d'un histogramme.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
import matplotlib.pyplot as plt

# Create a Spark session
spark = SparkSession.builder.appName("SalaryDistributionAnalysis").getOrCreate()

# Assuming 'result_df' is your PySpark DataFrame
# Specify the target column
target_column = "salary"

# Display summary statistics for the target variable
result_df.describe([target_column]).show()

# Visualize the distribution of the target variable (salary)
salary_distribution = result_df.select(target_column).toPandas()

plt.figure(figsize=(10, 6))
plt.hist(salary_distribution[target_column], bins=30, edgecolor='black')
plt.title('Distribution of Salaries')
plt.xlabel('Salary')
plt.ylabel('Frequency')
plt.show()
```

```
+-------+------------------+
|summary|            salary|
+-------+------------------+
|  count|           1000000|
|   mean|        116.061818|
| stddev|38.717936381133605|
|    min|                 0|
|    max|               301|
+-------+------------------+
```



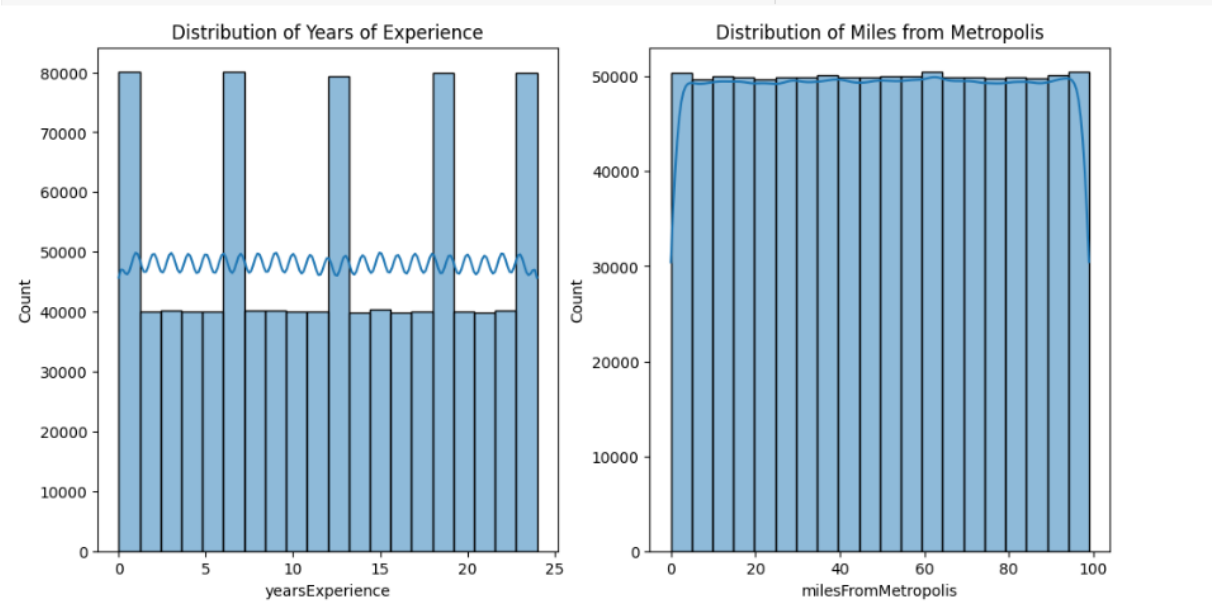
Interprétation : L'histogramme semble présenter une distribution en cloche, ce qui suggère une distribution normale des salaires. Cela implique que la majorité des salaires se regroupe autour de la moyenne, avec moins de fréquence aux extrémités basses et hautes on remarque aussi que les salaires sont répartis entre 0 et environ 300. D'après le visuel le pic le plus élevé de l'histogramme se situe autour de la valeur 100, ce qui pourrait indiquer que la

plupart des employés ont un salaire autour de cette valeur. En résumé La distribution semble être assez symétrique autour du pic, ce qui indique une variabilité uniforme des salaires au-dessus et en dessous de la moyenne.

2. la distribution des années d'expérience et de la distance depuis la métropole :

Le code suivant offre une visualisation graphique de la distribution des années d'expérience et de la distance depuis la métropole dans l'ensemble de données.

```
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

# Create a Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Assuming 'data' is your DataFrame
data = spark.read.csv('/content/train_dataset.csv', header=True, inferSchema=True)

# Convert PySpark DataFrame to Pandas for visualization
numeric_data_pd = data.select("yearsExperience", "milesFromMetropolis").toPandas()

# Visualization
plt.figure(figsize=(12, 6))

# Distribution of Years of Experience
plt.subplot(1, 2, 1)
sns.histplot(numeric_data_pd['yearsExperience'], bins=20, kde=True)
plt.title('Distribution of Years of Experience')

# Distribution of Miles from Metropolis
plt.subplot(1, 2, 2)
sns.histplot(numeric_data_pd['milesFromMetropolis'], bins=20, kde=True)
plt.title('Distribution of Miles from Metropolis')

plt.show()
```



Interprétation :

- **Distribution des années d'expérience :**

L'histogramme montre que la fréquence des années d'expérience est assez uniformément répartie, Il semble y avoir des augmentations périodiques à des intervalles réguliers. Cela

pourrait indiquer que les valeurs sont regroupées autour de nombres entiers, ce qui est logique pour les années d'expérience.

La ligne bleue claire en haut de chaque barre pourrait représenter la courbe de densité de kernel (KDE), mais elle semble être superposée directement sur les barres de l'histogramme, ce qui est inhabituel. Normalement, la courbe KDE serait plus lisse et suivrait la forme générale des données.

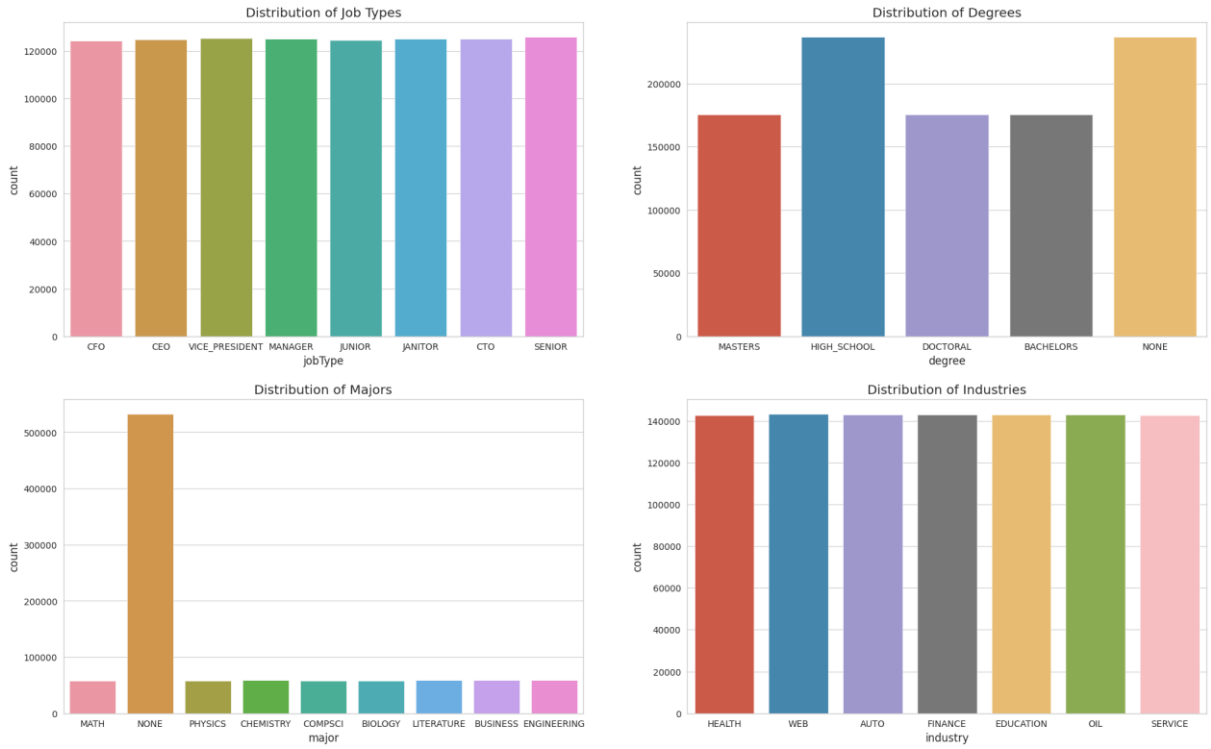
- **Distribution des miles depuis la métropole :**

Cette distribution est très différente de la première. Les fréquences sont presque identiques pour toutes les valeurs de distance, ce qui suggère que la localisation des employés par rapport à la métropole est uniformément répartie.

Comme pour le premier graphique, la ligne de KDE ne suit pas la tendance attendue d'une courbe lisse. Elle semble plate et superposée sur les barres, ce qui n'est pas représentatif de ce qu'on attend d'une courbe KDE.

3. La distribution des différentes variables catégorielles (Job Types, Degrees, Majors, Industries) :

```
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

# Create a Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Assuming 'data' is your DataFrame
data = spark.read.csv('/content/train_dataset.csv', header=True, inferSchema=True)

# Convert PySpark DataFrame to Pandas for visualization
categorical_data_pd = data.select("jobType", "degree", "major", "industry").toPandas()

# Visualization
plt.figure(figsize=(16, 8))

# Distribution of Job Types
plt.subplot(2, 2, 1)
sns.countplot(x="jobType", data=categorical_data_pd)
plt.title('Distribution of Job Types')

# Distribution of Degrees
plt.subplot(2, 2, 2)
sns.countplot(x="degree", data=categorical_data_pd)
plt.title('Distribution of Degrees')

# Distribution of Majors
plt.subplot(2, 2, 3)
sns.countplot(x="major", data=categorical_data_pd)
plt.title('Distribution of Majors')

# Distribution of Industries
plt.subplot(2, 2, 4)
sns.countplot(x="industry", data=categorical_data_pd)
plt.title('Distribution of Industries')

plt.show()
```



Interprétation :

Distribution des Types d'Emploi (Job Types) :

L'histogramme montre la répartition des types d'emploi. Les types d'emploi les plus fréquents semblent être "JUNIOR" et "SENIOR", tandis que les types d'emploi tels que "CEO" et "CFO" sont moins fréquents.

Distribution des Diplômes (Degrees) :

La deuxième visualisation montre la distribution des diplômes. On observe que la plupart des employés ont un diplôme de "HIGH_SCHOOL", suivi de "BACHELORS" et "MASTERS". Les diplômes de "DOCTORAL" sont moins fréquents.

Distribution des Spécialisations (Majors) :

L'histogramme des spécialisations montre la répartition des majors des employés. La plupart des employés n'ont pas de spécialisation ("NONE"), et parmi ceux qui en ont, "CHEMISTRY" et "PHYSICS" sont les plus courants.

Distribution des Secteurs Industriels (Industries) :

La dernière visualisation représente la distribution des secteurs industriels. Les secteurs tels que "HEALTH", "WEB", et "AUTO" semblent avoir un nombre significatif d'emplois, tandis que d'autres secteurs comme "EDUCATION" et "OIL" ont une fréquence relativement plus faible .

IV. Analyse et Transformation du Dataset :

1. Conversion des variables de type chaîne en entiers :

Le code suivant utilise PySpark pour appliquer l'indexation de chaînes (String Indexer) aux colonnes catégorielles .le résultat est un dataFrame où les colonnes catégorielles spécifiées ont été transformées en indices numériques à l'aide de l'indexeur de chaînes.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.sql import SparkSession

# Create a Spark session (if not already created)
spark = SparkSession.builder.appName("example").getOrCreate()

# List of categorical columns
categorical_columns = ["jobType", "degree", "major", "industry"]

# Apply StringIndexer to each categorical column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(result_df) for column in categorical_columns]

# Create a pipeline and fit it to the DataFrame
pipeline = Pipeline(stages=indexers)
indexed_df = pipeline.fit(result_df).transform(result_df)

# Show the transformed DataFrame
indexed_df.show()
```

jobId	companyId	jobType	degree	major	industry	yearsExperience	milesFromMetropolis	salary	jobType_index	degree_index	major_index	industry_index
JOB1362684407687	COMP37	CFO	MASTERS	MATH	HEALTH	10	83	130	7.0	4.0	8.0	5.0
JOB1362684407688	COMP19	CEO	HIGH_SCHOOL	NONE	WEB	3	73	101	5.0	0.0	0.0	0.0
JOB1362684407690	COMP38	MANAGER	DOCTORAL	CHEMISTRY	AUTO	8	17	142	2.0	3.0	1.0	1.0
JOB1362684407693	COMP15	CFO	NONE	NONE	HEALTH	23	24	178	7.0	1.0	0.0	5.0
JOB1362684407694	COMP24	JUNIOR	BACHELORS	CHEMISTRY	EDUCATION	9	70	73	6.0	2.0	1.0	3.0
JOB1362684407695	COMP20	JANITOR	HIGH_SCHOOL	NONE	EDUCATION	1	54	31	4.0	0.0	0.0	3.0
JOB1362684407696	COMP41	VICE_PRESIDENT	BACHELORS	CHEMISTRY	AUTO	17	68	104	1.0	2.0	1.0	1.0
JOB1362684407697	COMP56	JANITOR	HIGH_SCHOOL	NONE	HEALTH	24	30	102	4.0	0.0	0.0	5.0
JOB1362684407700	COMP54	JUNIOR	MASTERS	MATH	FINANCE	21	26	193	6.0	4.0	8.0	2.0
JOB1362684407703	COMP14	JUNIOR	MASTERS	PHYSICS	SERVICE	1	91	47	6.0	4.0	5.0	6.0
JOB1362684407705	COMP58	CEO	MASTERS	LITERATURE	SERVICE	23	66	122	5.0	4.0	2.0	6.0
JOB1362684407706	COMP3	CEO	MASTERS	PHYSICS	EDUCATION	9	99	95	5.0	4.0	5.0	3.0
JOB1362684407709	COMP30	JUNIOR	BACHELORS	LITERATURE	HEALTH	18	69	105	6.0	2.0	2.0	5.0
JOB1362684407710	COMP38	JUNIOR	NONE	NONE	HEALTH	20	63	76	6.0	1.0	0.0	5.0
JOB1362684407714	COMP34	CTO	MASTERS	BUSINESS	AUTO	9	6	130	3.0	4.0	4.0	1.0
JOB1362684407715	COMP11	JANITOR	HIGH_SCHOOL	NONE	FINANCE	20	23	101	4.0	0.0	0.0	2.0
JOB1362684407721	COMP31	MANAGER	DOCTORAL	NONE	HEALTH	15	14	164	2.0	3.0	0.0	5.0
JOB1362684407722	COMP47	MANAGER	NONE	NONE	HEALTH	18	58	115	2.0	1.0	0.0	5.0
JOB1362684407724	COMP8	VICE_PRESIDENT	DOCTORAL	BUSINESS	HEALTH	24	35	183	1.0	3.0	4.0	5.0
JOB1362684407725	COMP31	VICE_PRESIDENT	HIGH_SCHOOL	NONE	OIL	8	6	114	1.0	0.0	0.0	4.0

only showing top 20 rows

2. Vérification de l'équilibre du jeu de données ou non :

Le code effectue une vérification pour déterminer si le jeu de données est équilibré ou non, en se basant sur la colonne cible "salary".

```
# Import necessary libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Create a Spark session
spark = SparkSession.builder.appName("DatasetBalanceCheck").getOrCreate()

# Assuming 'result_df' is your PySpark DataFrame
# Specify the target column
target_column = "salary"

# Count the number of instances in each class
class_counts = result_df.groupBy(target_column).count()

# Check if there are instances with the specified value in the target column
positive_class_count = class_counts.filter(col(target_column) == 1).select("count").collect()

# Print the count if there are instances, otherwise print a message
if positive_class_count:
    positive_ratio = positive_class_count[0][0] / result_df.count()
    print("Positive class ratio:", positive_ratio)

    # Check if the positive class ratio is within a reasonable range for a balanced dataset
    if positive_ratio >= 0.4 and positive_ratio <= 0.6:
        print("Dataset is balanced")
    else:
        print("Dataset is unbalanced")
else:
    print(f"No instances with {target_column} equal to 1 in the dataset.")
```

```
No instances with salary equal to 1 in the dataset.
```

Interprétation : Le message "Aucune instance avec le salaire égal à 1 dans le jeu de données." indique qu'il n'y a aucune occurrence où la valeur de la colonne "salary" est égale à 1. cela peut signifier que la classe positive (classe représentée par la valeur 1 dans la colonne "salary") n'est pas présente dans le jeu de données ou qu'elle est très rare.

3. Normalisation des données :

Le code suivant utilise PySpark pour normaliser les données en appliquant l'indexation de chaînes (StringIndexer) aux colonnes catégorielles, puis en utilisant VectorAssembler pour combiner les colonnes catégorielles indexées en une seule colonne de vecteurs, et enfin, appliquant le MinMaxScaler pour normaliser ces vecteurs.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, MinMaxScaler
from pyspark.sql import SparkSession

# Create a Spark session (if not already created)
spark = SparkSession.builder.appName("example").getOrCreate()

# List of categorical columns
categorical_columns = ["industry"]

# Apply StringIndexer to each categorical column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(result_df) for column in categorical_columns]

# Use VectorAssembler to combine indexed categorical columns into a single vector column
assembler = VectorAssembler(inputCols=[column+"_index" for column in categorical_columns], outputCol="indexed_features")

# Apply MinMaxScaler to the vector column
scaler = MinMaxScaler(inputCol="indexed_features", outputCol="scaled_features")

# Create a pipeline and fit it to the DataFrame
pipeline = Pipeline(stages=indexers + [assembler, scaler])
normalized_df = pipeline.fit(result_df).transform(result_df)

# Show the transformed DataFrame
normalized_df.show()
```

```
+---------------+--------+-------------+----------+---------+--------+--------------+-----------------+------+----------+--------------+--------------------+
|          jobId|companyId|      jobType|    degree|    major|industry|yearsExperience|milesFromMetropolis|salary|industry_index|indexed_features|     scaled_features|
+---------------+--------+-------------+----------+---------+--------+--------------+-----------------+------+----------+--------------+--------------------+
|JOB1362684407687|  COMP37|          CFO|   MASTERS|     MATH|  HEALTH|            10|               83|   130|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407688|  COMP19|          CEO|HIGH_SCHOOL|     NONE|     WEB|             3|               73|   101|       0.0|         [0.0]|               [0.0]|
|JOB1362684407690|  COMP38|      MANAGER|  DOCTORAL|CHEMISTRY|    AUTO|             8|               17|   142|       1.0|         [1.0]|[0.16666666666666...|
|JOB1362684407693|  COMP15|          CFO|      NONE|     NONE|  HEALTH|            23|               24|   178|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407694|  COMP24|       JUNIOR|  BACHELORS|CHEMISTRY|EDUCATION|            9|               70|    73|       3.0|         [3.0]|               [0.5]|
|JOB1362684407695|  COMP20|      JANITOR|HIGH_SCHOOL|     NONE|EDUCATION|            1|               54|    31|       3.0|         [3.0]|               [0.5]|
|JOB1362684407696|  COMP41|VICE_PRESIDENT|  BACHELORS|CHEMISTRY|    AUTO|            17|               68|   104|       1.0|         [1.0]|[0.16666666666666...|
|JOB1362684407697|  COMP56|      JANITOR|HIGH_SCHOOL|     NONE|  HEALTH|            24|               30|   102|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407700|  COMP54|       JUNIOR|   MASTERS|     MATH| FINANCE|            21|               26|   193|       2.0|         [2.0]|[0.3333333333333333]|
|JOB1362684407703|  COMP14|       JUNIOR|   MASTERS|  PHYSICS| SERVICE|             1|               91|    47|       6.0|         [6.0]|               [1.0]|
|JOB1362684407705|  COMP58|          CEO|   MASTERS|LITERATURE| SERVICE|            23|               66|   122|       6.0|         [6.0]|               [1.0]|
|JOB1362684407706|   COMP3|          CEO|   MASTERS|  PHYSICS|EDUCATION|            9|               99|    95|       3.0|         [3.0]|               [0.5]|
|JOB1362684407709|  COMP30|       JUNIOR|  BACHELORS|LITERATURE|  HEALTH|            18|               69|   105|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407710|  COMP38|       JUNIOR|      NONE|     NONE|  HEALTH|            20|               63|    76|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407714|  COMP34|          CTO|   MASTERS| BUSINESS|    AUTO|             9|                6|   130|       1.0|         [1.0]|[0.16666666666666...|
|JOB1362684407715|  COMP11|      JANITOR|HIGH_SCHOOL|     NONE| FINANCE|            20|               23|   101|       2.0|         [2.0]|[0.3333333333333333]|
|JOB1362684407721|  COMP31|      MANAGER|  DOCTORAL|     NONE|  HEALTH|            15|               14|   164|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407722|  COMP47|      MANAGER|      NONE|     NONE|  HEALTH|            18|               58|   115|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407724|   COMP8|VICE_PRESIDENT|  DOCTORAL| BUSINESS|  HEALTH|            24|               35|   183|       5.0|         [5.0]|[0.8333333333333333]|
|JOB1362684407725|  COMP31|VICE_PRESIDENT|HIGH_SCHOOL|     NONE|     OIL|             8|                6|   114|       4.0|         [4.0]|[0.6666666666666666]|
+---------------+--------+-------------+----------+---------+--------+--------------+-----------------+------+----------+--------------+--------------------+
only showing top 20 rows
```

4. Corrélation entre les données :

Le code suivant calcule la matrice de corrélation entre les différents variables.

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler, MinMaxScaler
from pyspark.ml.stat import Correlation
from pyspark.sql import SparkSession

# Create a Spark session (if not already created)
spark = SparkSession.builder.appName("example").getOrCreate()

# List of categorical columns
categorical_columns = ["industry"]

# Apply StringIndexer to each categorical column
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(result_df) for column in categorical_columns]

# List of numerical columns
numerical_columns = ["yearsExperience", "milesFromMetropolis", "salary"]

# Use VectorAssembler to combine indexed categorical columns and numerical columns into a single vector column
assembler = VectorAssembler(inputCols=[column+"_index" for column in categorical_columns] + numerical_columns, outputCol="features")

# Apply MinMaxScaler to the vector column
scaler = MinMaxScaler(inputCol="features", outputCol="scaled_features")

# Create a pipeline and fit it to the DataFrame
pipeline = Pipeline(stages=indexers + [assembler, scaler])
normalized_df = pipeline.fit(result_df).transform(result_df)

# Calculate the correlation matrix
correlation_matrix = Correlation.corr(normalized_df, "scaled_features").head()

# Extract the correlation matrix as a NumPy array
corr_matrix_np = correlation_matrix[0].toArray()

# Print the correlation matrix
print("Correlation Matrix:")
print(corr_matrix_np)
```

```
Correlation Matrix:
[[ 1.00000000e+00  1.23968974e-04  4.87615662e-04 -7.15199177e-02]
 [ 1.23968974e-04  1.00000000e+00  6.72698342e-04  3.75012700e-01]
 [ 4.87615662e-04  6.72698342e-04  1.00000000e+00 -2.97666353e-01]
 [-7.15199177e-02  3.75012700e-01 -2.97666353e-01  1.00000000e+00]]
```

Interprétation : En examinant la matrice de corrélation fournie :

- **Corrélation entre "industry" et les autres variables :** la première ligne (correspondant à "industry") a des coefficients très proches de zéro avec les autres variables. Cela suggère une faible corrélation linéaire entre "industry" et les autres variables.
- **Corrélation entre "yearsExperience" et "salary" :** le coefficient de 0.375 suggère une corrélation positive modérée entre l'expérience en années et le salaire Cela signifie que, à mesure que l'expérience augmente, le salaire a tendance à augmenter.
- **Corrélation entre "milesFromMetropolis" et "salary" :** le coefficient de -0.297 suggère une corrélation négative modérée entre la distance depuis la métropole. Cela indique que, à mesure que la distance depuis la métropole augmente, le salaire a tendance à diminuer.
- **Corrélation entre les autres variables :** les coefficients entre "industry", "yearsExperience", et "milesFromMetropolis" sont très proches de zéro, indiquant une faible corrélation linéaire.

V. Application des Modèles ML en utilisant Spark :

1. Compréhension de la distribution des salaires pour une meilleure prédiction :

Le caractéristique "salaire" présente une distribution à queue longue asymétrique à droite, avec certains salaires qui apparaissent de nombreuses fois dans l'ensemble de données (les "pics" dans cet histogramme).

Interprétation : Les trois lignes verticales sont tracées pour indiquer la moyenne (rouge), la médiane (violet) et le mode (vert) des salaires.

- La moyenne est indiquée à environ 116 100 $ (rouge).
- La médiane est légèrement inférieure à la moyenne, indiquée à environ 114 000 $ (violet), ce qui est typique des distributions asymétriques où la moyenne est influencée par les valeurs extrêmes.
- Le mode est le salaire le plus fréquent et est indiqué à environ 108 000 $ (vert).

2. Encodage des Labels :

Le script effectue le prétraitement de données pour ML en utilisant PySpark. Il commence par indexer les colonnes catégorielles comme "companyId", "jobType", et d'autres, les transformant en indices numériques pour les rendre compatibles avec les modèles de ML. Ensuite, il combine ces indices avec des colonnes numériques existantes dans un vecteur unique de caractéristiques. L'encodage One-Hot est également appliqué aux variables catégorielles pour éviter les relations ordinales indésirables. Ces étapes sont regroupées dans un Pipeline pour faciliter l'application et la gestion du traitement des données, rendant le processus plus structuré et efficace, Il est préférable d'encoder la variable 'degree' car c'est la seule variable catégorielle qui présente une hiérarchie

```
[ ] categoricalColumns = ["companyId","jobType", "degree", "major", "industry"]
    stages = []
    for categoricalCol in categoricalColumns:
        stringIndexer = StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol + 'Index')
        stages += [stringIndexer]
    numericCols =  ["yearsExperience", "milesFromMetropolis"]
    assemblerInputs = [c + "Index" for c in categoricalColumns] + numericCols
    assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
    stages += [assembler]
```

```
stages = []
stringIndexer = StringIndexer(inputCol = "degree", outputCol = "degreeIndex")
stages += [stringIndexer]
```

```
[ ] categoricalColumns = ["companyId","jobType", "major", "industry"]
    for categoricalCol in categoricalColumns:
        stringIndexer = StringIndexer(inputCol = categoricalCol, outputCol = categoricalCol + 'Index')
        encoder = OneHotEncoder(inputCols=[stringIndexer.getOutputCol()], outputCols=[categoricalCol + "classVec"])
        stages += [stringIndexer, encoder]
    numericCols =  ["yearsExperience", "milesFromMetropolis"]
    assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols + ["degreeIndex"]
    assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
    stages += [assembler]
    pipeline = Pipeline(stages = stages)
    pipelineModel = pipeline.fit(df_encoded)
    df_encoded = pipelineModel.transform(df_encoded)
    selectedCols = ['features'] + cols
    df_encoded = df_encoded.select(selectedCols)
```

3. Division des données en ensembles d'entraînement et de test :

```
train_data, test_data = df_encoded.randomSplit([0.995, 0.005], seed = 42)
print("There are %d training examples and %d test examples." % (train_data.count(), test_data.count()))
```

```
There are 994977 training examples and 5023 test examples.
```

4. Application du 1 er Modèle : Gradient Boosting Trees :

- Création d'une instance de **l'algorithme GBT** pour la régression :

L'application du modèle Gradient Boosting Trees (GBT) sur l'ensemble de données de salaires d'employés permet de prédire et d'analyser les salaires des employés. Ce modèle peut être utilisé pour estimer les salaires des employés en fonction de différentes variables telles que l'expérience, l'éducation et le poste, ce qui aide à évaluer les niveaux de rémunération, à identifier les facteurs clés influençant les salaires, et à prendre des décisions éclairées en matière de rémunération.

```
[ ] gbt =  GBTRegressor(featuresCol="features", labelCol="salary", maxBins=20, maxDepth=12)
    gbt_model = gbt.fit(train_data)
    predictions = gbt_model.transform(test_data)
```

- Evaluation des Résultats du modèle :

Ce code calcule la racine de **l'erreur quadratique moyenne (RMSE)** pour évaluer la précision du modèle.il mesure la différence moyenne entre les prédictions et les valeurs réelles, en prenant en compte les carrés de ces différences pour donner une indication de la précision du modèle. Plus le RMSE est bas, plus le modèle est précis dans ses prédictions. Cette évaluation permet de quantifier la performance du modèle dans la prédiction des salaires des employés en termes d'erreur moyenne.

```
evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
rmse
```

```
19.315359569894024
```

Interprétation : D'après l'observation l'échelle des salaires dans l'ensemble de données on remarque que l'éventail des salaires est assez large (par exemple ici salaires varient de 40k$ à 200k$) donc un RMSE de 19.49 K$ peut être considéré comme une valeur moyenne, indiquant que le modèle fait des prédictions moyennes à précises.

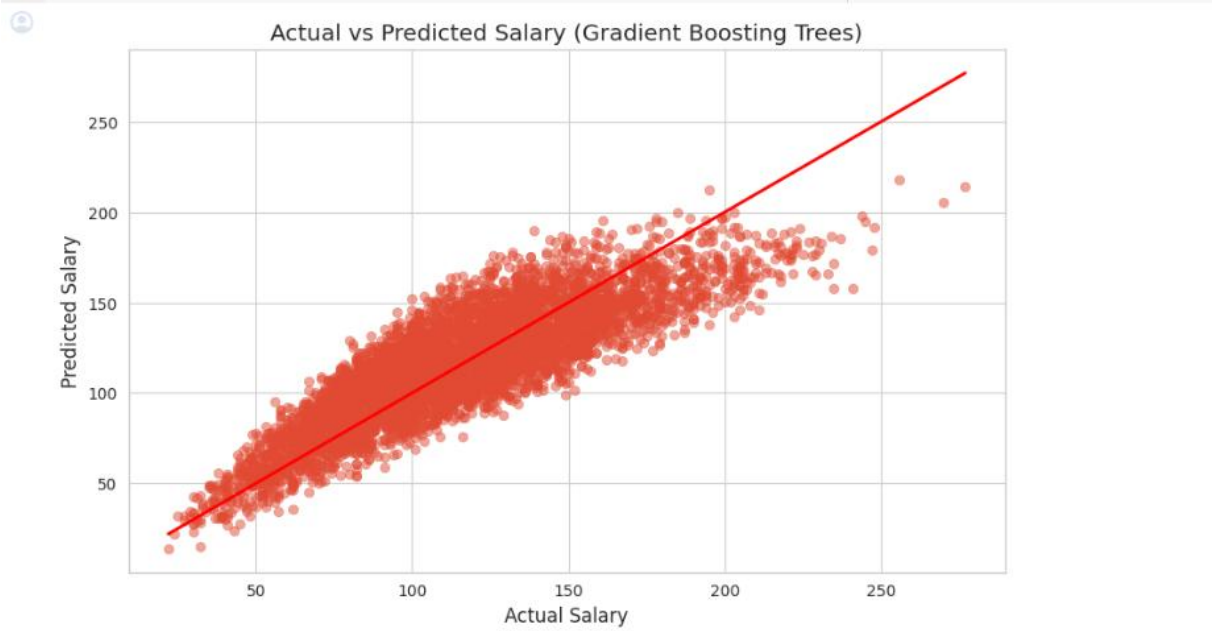
Ces résultats confirment que le salaire d'un employé dépend fortement de type de poste occupé par cet employé par exemple le salaire d'un CEO n'est pas le même que celui d'un MANAGER. Il dépend aussi du niveau de diplôme obtenu par l'employé ainsi que la spécialité de l'étude sans oublier le secteur d'activité de l'entreprise.

Le code utilise l'indice **R-carré (R2)** pour évaluer la performance du modèle. Il commence par créer un objet "evaluator" de type RegressionEvaluator, configuré pour mesurer l'indice R2. L'indice R2 quantifie la proportion de la variance totale des valeurs de la variable cible (dans ce cas, les salaires des employés) expliquée par le modèle. Plus précisément, il mesure à quel point les prédictions du modèle sont proches des valeurs réelles.

```
[ ] evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="r2")
    r2 = evaluator.evaluate(predictions)
    r2

    0.7464470532595058
```

Interprétation : Un indice R2 de 0.746 est généralement considéré comme une bonne performance pour un modèle de régression, car il indique que le modèle parvient à expliquer une grande proportion de la variabilité des données.

- Comparaison entre le salaire réel et prédits :



Interprétation : Le scatter plot montre que le modèle Gradient Boosting Trees a une bonne performance, avec la majorité des prédictions proches de la ligne de parité (ligne rouge).

Cependant, il y a une dispersion des points, surtout aux extrémités de l'échelle salariale, indiquant des erreurs de prédiction plus importantes pour les salaires très bas ou très élevés.

La présence de certains points éloignés de la ligne rouge, notamment dans la gamme de salaires élevés, pourrait indiquer des valeurs aberrantes ou des cas où le modèle ne prédit pas aussi bien.

5. Application du 2 ème Modèle : Régression linéaire

L'application d'un modèle de régression linéaire sur un ensemble de données de salaires d'employés est un outil essentiel pour comprendre les relations entre les caractéristiques des employés et leurs salaires, ainsi que pour générer des prédictions basées sur ces relations. Cela peut aider les entreprises à prendre des décisions éclairées en matière de rémunération, de gestion des ressources humaines et d'optimisation de la politique salariale.

```
from pyspark.ml.regression import LinearRegression
# Modèle de régression linéaire
lr = LinearRegression(featuresCol='features', labelCol='salary')
# Entraînement du modèle
model = lr.fit(train_data)
# Prédictions
predictions1 = model.transform(test_data)
```

- Evaluation des Résultats du modèle :

Ce code réalise une évaluation complète du modèle de régression en utilisant deux métriques couramment utilisées : la racine de l'erreur quadratique moyenne (RMSE) et l'indice R2.

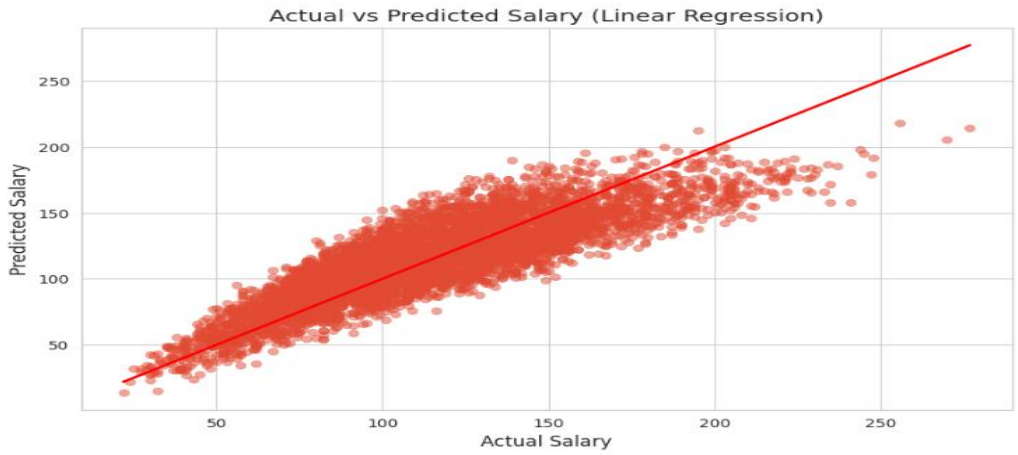
```
# Évaluation du modèle
evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions1)
print("RMSE: %f" % rmse)

evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="r2")
r2 = evaluator.evaluate(predictions1)
print("R2: %f" % r2)
```

```
RMSE: 19.952386
R2: 0.729447
```

Interprétation :

- RMSE : Environ 19.95. Cela signifie que, en moyenne, les prédictions du modèle ont une erreur d'environ 19.95 unités par rapport aux vraies valeurs des salaires des employés. Une RMSE plus basse indiquerait une meilleure adéquation du modèle aux données, mais une RMSE de 19.95 suggère déjà une performance relativement décente.
- R2 : Environ 0.729. Cela signifie que le modèle explique environ 72.9% de la variance totale des salaires des employés dans les données, ce qui est généralement considéré comme un bon résultat. Un R2 de 0.729 suggère que le modèle capture une grande partie de la variabilité des données.

- Comparaison entre le salaire réel et prédits :



Interprétation : Le modèle semble fournir de bonnes prédictions pour une grande partie des données, mais il pourrait être amélioré, en particulier pour prédire avec précision les cas extrêmes. Il est possible que le modèle bénéficie d'un ajustement ou de la prise en compte de caractéristiques supplémentaires pour mieux capter la complexité des données de salaire dans **ces gammes.**

6. Application du 3 ème Modèle : Random Forest

L'application d'un modèle Random Forest sur un ensemble de données de salaires d'employés permet d'obtenir des prédictions précises, de modéliser des relations complexes, d'éviter le surajustement, d'identifier les caractéristiques importantes, et de fournir une alternative robuste aux autres modèles de régression. Cela peut être utile pour une meilleure compréhension des facteurs qui influent sur les salaires des employés et pour améliorer les prédictions dans le contexte de gestion des ressources humaines et de rémunération.

```
[ ] from pyspark.ml.regression import RandomForestRegressor, GBTRegressor
    # Pour les Forêts Aléatoires
    rf = RandomForestRegressor(featuresCol='features', labelCol='salary')
    #Entraînez le modèle
    model = rf.fit(train_data)
    # Application des prédictions
    predictions2= model.transform(test_data)
```

- Evaluation des Résultats du modèle :

Ce code effectue une évaluation du modèle Random Forest en utilisant deux métriques couramment utilisées : la racine de l'erreur quadratique moyenne (RMSE) et l'indice R2.

```
# Évaluation du modèle
evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions2)
print("RMSE: %f" % rmse)

evaluator = RegressionEvaluator(labelCol="salary", predictionCol="prediction", metricName="r2")
r2 = evaluator.evaluate(predictions2)
print("R2: %f" % r2)
```

```
RMSE: 25.753485
R2: 0.549251
```

Interprétation : Ces résultats indiquent que le modèle Random Forest a une performance modérée pour prédire les salaires des employés en fonction des caractéristiques spécifiques.

- RMSE (Racine de l'erreur quadratique moyenne) : Environ 25.75. Cela signifie que, en moyenne, les prédictions du modèle Random Forest ont une erreur d'environ 25.75 unités par rapport aux vraies valeurs des salaires des employés. Une RMSE plus basse aurait indiqué une meilleure adéquation du modèle aux données, mais une RMSE de 25.75 donne toujours une indication de la performance du modèle.
- R2 (indice R-carré) : Environ 0.549. Cela signifie que le modèle Random Forest explique environ 54.9% de la variance totale des salaires des employés dans les données. Un R2 de 0.549 indique que le modèle capture une proportion importante de la variabilité des données.

- Comparaison entre le salaire réel et prédits :

Interprétation :

La majorité des points semblent être regroupés autour de la ligne, indiquant que les prédictions sont relativement précises pour la plupart des données. Cependant, il semble y avoir une certaine variabilité, en particulier pour les salaires plus élevés, où le modèle a tendance à sous-estimer ou surestimer les salaires réels. De plus, il y a quelques valeurs aberrantes visibles qui s'écartent significativement de la ligne rouge.

Cela peut suggérer que le modèle de forêt aléatoire fonctionne bien pour une gamme de données, mais il pourrait être amélioré pour augmenter la précision des prédictions, en particulier dans les extrêmes supérieurs de l'échelle des salaires. Il serait utile d'examiner de

plus près les cas où il y a de grandes déviations pour comprendre pourquoi le modèle n'a pas bien performé et pour voir s'il y a des caractéristiques supplémentaires ou des ajustements de paramètres qui pourraient améliorer les prédictions.

7. Comparaison entre les trois modèles :

Après avoir examiné les trois graphiques de dispersion pour les modèles de Gradient Boosting Trees, de régression linéaire et de forêt aléatoire (Random Forest), voici une conclusion globale sur la performance de ces modèles de prédiction de salaire des employés :

- **Gradient Boosting Trees** : Les points se concentraient étroitement autour de la ligne rouge, indiquant que les prédictions étaient assez précises sur toute la gamme de salaires. Cependant, il y avait des écarts plus significatifs aux extrémités de l'échelle des salaires, en particulier pour les salaires plus élevés.
- **Régression linéaire :** La distribution des points suggérait que le modèle avait une bonne performance générale mais semblait moins précis que le Gradient Boosting Trees pour les valeurs extrêmes. Les points étaient légèrement plus dispersés autour de la ligne rouge, indiquant une variance plus élevée dans les erreurs de prédiction.
- **Forêt aléatoire (Random Forest)** : Les points étaient également proches de la ligne rouge, montrant une bonne adéquation des prédictions aux valeurs réelles, similaire à celle du Gradient Boosting Trees. Toutefois, il y avait également une tendance à la sous-estimation ou à la surestimation pour les salaires plus élevés, et quelques valeurs aberrantes étaient présentes.

En conclusion, les trois modèles ont montré une capacité à prédire les salaires de manière relativement précise, avec chacun ayant des forces et des faiblesses distinctes. Le modèle de Gradient Boosting Trees semblait avoir une légère avance en termes de précision globale, bien que tous les modèles aient du mal avec les valeurs extrêmes, ce qui est courant dans les problèmes de régression. Les valeurs aberrantes et les prédictions inexactes aux extrémités de l'échelle des salaires indiquent que des améliorations pourraient être apportées, peut-être en affinant davantage les paramètres du modèle, en utilisant des techniques d'ensemble, ou en explorant des caractéristiques supplémentaires qui pourraient améliorer la prédiction pour ces cas.

CONCLUSION :

Au cours de ce projet, nous avons exploré trois modèles différents pour prédire les salaires des employés en utilisant Apache Spark : la régression linéaire, les Gradient Boosting Trees (GBT) et les Forêts Aléatoires (Random Forest). Chacun de ces modèles a ses propres forces et peut être adapté à différents types de données et à la complexité des relations entre les caractéristiques et la variable cible, le salaire.

Les Gradient Boosting Trees se sont révélés être le meilleur modèle parmi ceux que nous avons testés, offrant un équilibre optimal entre précision et capacité à capturer la complexité des données. Leur capacité à intégrer une grande variété de facteurs liés au travail, tels que le type d'emploi, le niveau d'éducation, la spécialité et l'industrie, a permis de fournir des estimations détaillées des salaires.

Apache Spark, avec ses outils MLlib, a joué un rôle crucial en permettant une application efficace de ces modèles sur de grands ensembles de données de manière distribuée. Cela a rendu l'analyse rapide et gérable même pour des calculs complexes, facilitant la gestion des différentes étapes du processus de ML, de la préparation des données à l'évaluation des modèles, le tout dans un environnement unifié.

Enfin, l'application de ces modèles de ML dans Spark a permis de mettre en lumière comment les salaires sont influencés par divers facteurs liés au travail, ce qui aide les organisations à élaborer des stratégies de rémunération éclairées et à identifier les domaines d'investissement en capital humain les

plus rentables. La capacité à prédire les salaires avec une telle précision constitue une compétence précieuse pour optimiser la gestion des ressources humaines et pour la planification stratégique au sein des entreprises.