

Rapport projet : Analyse Comportementale et Recommandation Produit sur un Site E-commerce avec Spark

Présenté par :

- mouhamadou diouf cissé
- ndeye fatou niassy

plan du projet

1. Introduction
2. Contexte du projet
3. Objectifs du projet
4. Analyse exploratoire des données
5. Prétraitement des données

1. Présentation du Projet

Contexte

Le présent projet vise à concevoir une solution d'analyse comportementale des utilisateurs d'une plateforme e-commerce, avec pour finalité l'élaboration d'un système de recommandation intelligent. Ce projet, mené dans un cadre académique, s'appuie sur des technologies Big Data et des approches d'intelligence artificielle afin de traiter efficacement des volumes importants de données générés par les utilisateurs.

Objectifs du Projet

Le projet vise à concevoir une solution complète d'analyse comportementale et de recommandation intelligente dans un contexte e-commerce. Pour cela, plusieurs objectifs principaux ont été définis :

- **Compréhension et modélisation des comportements utilisateurs**

L'objectif initial est d'analyser en profondeur les comportements de navigation et d'achat des utilisateurs à partir des données de logs collectées sur une plateforme e-commerce. Il s'agit notamment d'identifier les actions réalisées par les utilisateurs (consultations, ajouts au panier, achats), d'étudier la chronologie et la fréquence de ces actions, et d'en extraire des tendances comportementales significatives. Ces analyses doivent permettre d'établir un modèle représentatif des interactions typiques entre un utilisateur et la plateforme.

- **Identification des patterns d’interactions et des profils types**

À partir des données collectées, il est essentiel d’identifier des schémas d’interactions récurrents, appelés patterns, permettant de regrouper les utilisateurs selon des comportements similaires. Cette étape vise à faire émerger des profils types de clients (par exemple : acheteurs impulsifs, visiteurs réguliers non acheteurs, utilisateurs sensibles au prix, etc.), facilitant ainsi une segmentation fine de la clientèle et une personnalisation plus pertinente des actions marketing ou commerciales.

- **Mise en place d’un système de recommandation dynamique et personnalisé**

Sur la base des préférences observées (historique de navigation, fréquence d’achat, catégories préférées, marques consultées), un système de recommandation doit être développé. Ce système devra être capable de proposer de manière dynamique des produits pertinents à chaque utilisateur, en tenant compte à la fois de ses habitudes passées et de son profil comportemental. Les recommandations devront s’appuyer sur des techniques hybrides combinant filtrage collaboratif, analyse de contenu, et segmentation client.

- **Simulation d’un environnement temps réel pour évaluation des performances**

Enfin, une infrastructure de simulation en temps réel sera mise en place afin de tester la robustesse et la réactivité de l’ensemble du système. Cette simulation permettra de reproduire l’arrivée continue des données, de déclencher le traitement de ces flux par les modules analytiques et de mesurer les performances en conditions quasi-réelles.

Données

Les jeux de données fournis sont au format CSV et couvrent les événements de navigation et d’achat sur les mois d’octobre et novembre. Chaque enregistrement contient des informations sur le type d’événement, les produits concernés, les catégories, la marque, le prix, l’utilisateur et sa session. Le volume de données est estimé à plusieurs dizaines de millions de lignes, ce qui justifie l’adoption d’une architecture Big Data.

2. Analyse exploratoire des données

Aperçu des données

Échantillon des premières lignes du jeu de données :

event_time	event_type	product_id	category_id	category_code	brand	price	user_id	user_session
2019-10-01 00:00:00	view	44600021	10380745	950138	7724 shiseido	35.79	5413127240	724076fde-8bb3-4e0...
2019-10-01 00:00:00	view	39008220	53013552	39017500	55 environment	33.20	554748313	8313dfbd-b87a-470...
2019-10-01 00:00:01	view	17200520	53013559	79203247	living NUll	543.10	191073365	736511c2-e2e3-422...
2019-10-01 00:00:01	view	13070620	53013558	92091749	inditelov	251.74	500508390	fc70-0e80-459...
2019-10-01 00:00:04	view	10042320	53013556	31882655	smapple	1081.98	58713517	451d7419-2748-4c5...

Remarque : seules les 5 premières lignes sont affichées.

Répartition des types d'événements :

event_type	count
view	40,779,399
cart	926,516
purchase	742,849

Le jeu de données contient plusieurs colonnes clés relatives au comportement des utilisateurs sur une plateforme e-commerce :

- *event_time* : Date et heure précises de l'événement.
- *event_type* : Type d'action effectuée par l'utilisateur (view, cart, purchase).
- *product_id* : Identifiant unique du produit concerné.
- *category_id* et *category_code* : Informations sur la catégorie du produit.
- *brand* : Marque du produit (des valeurs peuvent être manquantes).
- *price* : Prix du produit.
- *user_id* : Identifiant de l'utilisateur.
- *user_session* : Identifiant unique de la session utilisateur (utile pour reconstituer les parcours utilisateurs).

Analyse Exploratoire des Données d'Événements eCommerce

Aperçu des Données

event_time	event_type	product_id	category_id	category_code	brand	price	user_id	user_session
2019-10-01 00:00:00	view	446000621	0380745	959113	7724 shiseido	35.79	541312720	76fde-8bb3-4e0...
2019-10-01 00:00:00	view	3900822	05301355	2326770995	enquana	33.2	554748933	3dfbd-b87a-470...
2019-10-01 00:00:01	view	17200520	05301355	9792632471	living_Urlo	543.1	519107360	511c2-e2e3-422...
2019-10-01 00:00:01	view	1307062	05301355	8920217491	notebook	251.74	550050859	0fc70-0e80-459...
2019-10-01 00:00:04	view	1004232	05301355	5631882655	smapple	1081.98	535871261	7d7419-2748-4c5...

Affichage des 5 premières lignes du dataset.

Analyse

- *View* est de loin l'événement le plus fréquent, représentant environ 96.1% de toutes les interactions.
- Les événements *cart* (ajouts au panier) sont environ *2.2%, tandis que les ***purchases** (achats) ne représentent que 1.7%.
- Ce déséquilibre est *typique dans les données eCommerce* : la majorité des utilisateurs consultent les produits sans nécessairement acheter.

Visualisation de la Distribution des Événements

Histogramme et Camembert

Figure : Diagramme en barres et camembert représentant la distribution et la proportion des événements (*view*, *cart*, *purchase*).

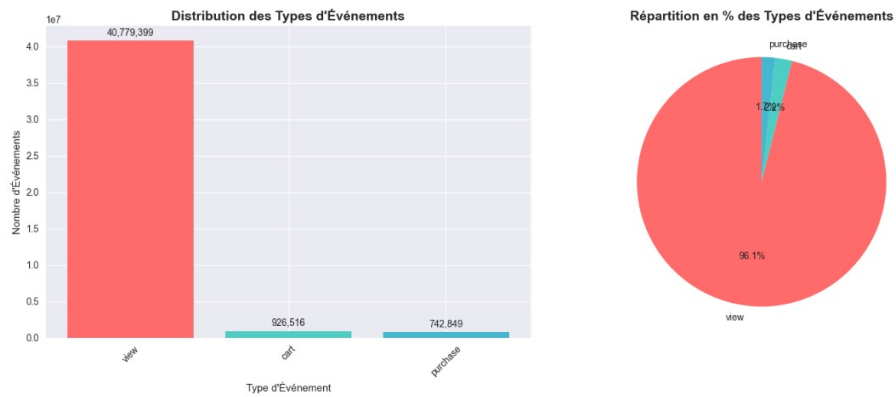


Figure 1: image.jpg

Interprétation

- *Taux de conversion faible* : Seuls une faible proportion des utilisateurs passent du view au purchase.
- *Potentiel d'optimisation* :
 - Travailler sur la rétention des paniers (cart abandonment).
 - Analyser les produits les plus vus vs. ceux les plus achetés.
 - Améliorer le parcours utilisateur pour favoriser l'achat

Analyse des Valeurs Manquantes

L'examen des valeurs manquantes permet de mieux comprendre la qualité et la complétude des données avant tout traitement ou modélisation.

Tableau des Valeurs Manquantes

Colonne	Valeurs Manquantes	Pourcentage (%)
category_code	13,515,609	31.84%
brand	6,113,008	14.40%
user_session	2	0.000005%
event_time	0	0.00%
event_type	0	0.00%
product_id	0	0.00%
category_id	0	0.00%
price	0	0.00%
user_id	0	0.00%

Interprétation

- *Colonnes sans valeurs manquantes* : event_time, event_type, product_id, category_id, price, et user_id. Ces colonnes sont entièrement complètes.
- *Colonnes avec valeurs manquantes notables* :
 - category_code : 31.84% de valeurs manquantes, ce qui est élevé. Cela peut poser problème pour la catégorisation ou les analyses liées à la classification de produits.
 - brand : 14.40% de valeurs manquantes. Cela pourrait affecter les analyses de fidélité à la marque ou les études de préférences.
- *Colonne avec peu de valeurs manquantes* : user_session ne contient que 2 valeurs manquantes, ce qui est négligeable et facilement nettoyable.

Analyse Temporelle

L'analyse temporelle permet de comprendre quand les utilisateurs interagissent le plus avec la plateforme, en examinant les tendances par *jour* et par *heure*.

Distribution des Événements par Jour du Mois (Octobre 2019)

Jour du Mois	Nombre d'Événements
1	1 244 245
2	1 191 328
3	1 127 303
4	1 417 190
5	1 330 339
6	1 318 379
7	1 200 531
8	1 370 633
9	1 347 543
10	1 284 077

Observation : Les événements sont globalement bien répartis sur le mois d'octobre, avec un pic autour du *15 au 20 octobre*. Il n'y a pas de chute brutale, ce qui suggère une activité régulière des utilisateurs.

Distribution des Événements par Heure de la Journée

Heure	Nombre d'Événements
0	306 805
1	559 027
2	1 069 047

Heure	Nombre d'Événements
3	1 550 285
4	1 915 643
5	2 125 633
6	2 269 092
7	2 335 718
8	2 390 127
9	2 351 683

Observation :

- L'activité commence à monter à partir de *3h du matin, atteint son pic entre **8h et 15h, *puis redescend progressivement*. - Le pic horaire* se situe vers 15h, suggérant un comportement d'achat ou de navigation intensif en journée.

Analyse des Prix

L'analyse des prix permet de comparer les comportements des utilisateurs selon le type d'événement (vue, ajout au panier, achat), en se basant sur les statistiques suivantes : prix moyen, minimum, maximum et le nombre de valeurs non nulles.

Statistiques des Prix par Type d'Événement

Type d'Événement	Prix Moyen (en \$)	Prix Min	Prix Max	Valeurs Non Nulles
Purchase	309.56	0.77	2 574.07	742 849
View	288.98	0.00	2 574.07	40 779 399
Cart	333.83	0.00	2 574.04	926 516

Observations

- *Prix Moyen :*
 - Les produits ajoutés au panier (*cart*) ont un prix moyen plus élevé (333,83 \$) que ceux seulement consultés (288,98 \$) ou achetés (309,56 \$).
 - Cela peut indiquer une intention d'achat pour des produits de plus grande valeur qui ne sont pas toujours concrétisée.
- *Prix Min = 0.00 :*
 - Le prix minimal est nul pour les événements view et cart, ce qui peut signaler la présence d'articles gratuits ou une donnée erronée.
- *Prix Max similaire :*
 - Le prix maximal est très proche pour tous les types d'événements (2 574 \$), ce qui montre qu'il existe des produits haut de gamme

qui suscitent à la fois de la consultation, de l'ajout au panier et des achats.

- *Volume de données :*
 - Les événements de type view sont les plus nombreux, ce qui est logique car les utilisateurs consultent bien plus qu'ils n'achètent ou ajoutent au panier.

Analyse des Sessions

L'analyse des sessions permet de mieux comprendre le comportement utilisateur sur la plateforme, en mesurant le nombre d'événements par session et la durée des sessions.

Distribution du Nombre d'Événements par Session

Statistique	Valeur
Nombre total	9 244 422
Moyenne	4.59 événements/session
Écart-type	6.77
Minimum	1 événement
25e percentile	1 événement
Médiane (50e)	2 événements
75e percentile	5 événements
Maximum	1 159 événements

Durée des Sessions

- *Durée moyenne :* 17,37 minutes
- *Durée médiane :* 1,05 minute

Observations

- *Sessions courtes :*
 - La médiane très basse (1,05 minute) indique que la majorité des sessions sont *très brèves*, avec peu d'interactions.
 - Cela suggère une navigation rapide ou peu engageante pour de nombreux utilisateurs.
- *Sessions longues mais rares :*
 - La valeur maximale de *1 159 événements* montre qu'il existe des sessions très actives, mais ce sont des *cas exceptionnels*.
- *Comportement utilisateur :*
 - 75 % des sessions contiennent *5 événements ou moins*, ce qui renforce l'idée que la plateforme est souvent utilisée de manière ponctuelle.

- Ces statistiques peuvent être utiles pour *segmenter les utilisateurs* selon leur engagement ou ajuster l'interface utilisateur (UX).

3. Prétraitement des données

Tâches de prétraitement réalisées

La fonction preprocess_data effectue plusieurs opérations de prétraitement sur les données brutes. Voici les étapes principales :

1. Extraction des caractéristiques temporelles et nettoyage des donnée

À partir de la colonne event_time, plusieurs nouvelles colonnes sont créées pour enrichir les données temporelles :

- hour : heure de l'événement
- minute : minute de l'événement
- second : seconde de l'événement
- day : jour du mois
- month : mois
- dayofweek : jour de la semaine
- date : date au format yyyy-MM-dd
- hour_bucket : regroupement horaire (date et heure arrondie à l'heure, format yyyy-MM-dd HH:00:00)

event_time	product_id	category_id	brand_id	user_id	session_id	minute	second	day	month	year	hour_bucket	
2019-10-01 00:00:00	view446000	003807	145055	387729	1372170	fd0	0	0	1	10	3	2019-10-01 00:00:00
2019-10-01 00:00:00	view390025	130135	123267	709255	1793737	fb0	0	0	1	10	3	2019-10-01 00:00:00
2019-10-01 00:00:01	view172000	003807	145055	387729	1372170	fd0	0	1	1	10	3	2019-10-01 00:00:01
2019-10-01 00:00:01	view130700	130135	123267	709255	1793737	fb0	0	1	1	10	3	2019-10-01 00:00:01

event	event_type	category_code	brand	price	user_id	session_id	min	second	day	month	day_of_week	hour	bucket
2019-10-01 00:00:04	view	1004205301	Electronics	1355560184965798702	17419-0	4	1	10	3	2019	2019-10-01	00:00:00	

2. Traitement des valeurs manquantes

Pour garantir la qualité des données, certains champs catégoriels sont nettoyés :

- category_code :
 - Remplacement des valeurs null ou “NaN” par “unknown”
- brand :
 - Remplacement des valeurs null par “unknown”

3. Nettoyage des prix

- La colonne price est corrigée en remplaçant les valeurs null ou inférieures ou égales à 0 par null (valeur manquante) # Entraînement des Modèles d’IA avec Apache Spark

Vue d’ensemble

Cette section présente l’implémentation de l’entraînement des modèles d’intelligence artificielle pour le système de recommandation e-commerce utilisant Apache Spark. Le processus comprend deux composants principaux : un modèle de segmentation client basé sur l’approche RFM (Recency, Frequency, Monetary) et un système de recommandation collaborative utilisant l’algorithme ALS (Alternating Least Squares).

Architecture du système d’entraînement

Classe principale : EcommerceModelTrainer

Le système d’entraînement est encapsulé dans la classe **EcommerceModelTrainer** qui orchestre l’ensemble du processus. Cette classe centralise la gestion de la session Spark, la préparation des données et l’entraînement des modèles.

Configuration Spark

La méthode `_create_spark_session()` initialise la session Spark avec une configuration optimisée :

```
SparkSession.builder \
    .appName("E-commerce Model Training") \
    .config("spark.driver.memory", "8g") \
    .config("spark.executor.memory", "8g") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") \
    .master("local[*]")
```

Cette configuration alloue 8 Go de mémoire aux processus driver et executor, active l'optimisation adaptative des requêtes SQL et utilise la sérialisation Kryo pour améliorer les performances.

Fonctions de préparation des données

prepare_user_features()

But : Calculer les caractéristiques comportementales et RFM pour chaque utilisateur

Cette fonction transforme les données brutes d'événements en métriques utilisateur exploitables :

```
# Calcul de la date de référence (date maximale dans les données)
max_date = cleaned_df.select(max("event_time")).first()[0]

# Calcul des métriques RFM et comportementales
user_features = cleaned_df.groupBy("user_id").agg(
    # Recency: nombre de jours depuis la dernière activité
    datediff(lit(max_date), max("event_time")).alias("recency"),

    # Frequency: nombre de jours distincts d'activité
    countDistinct(col("event_time").cast("date")).alias("frequency"),

    # Monetary: montant total des achats
    sum(when(col("event_type") == "purchase", col("price")).otherwise(0)).alias("monetary"),

    # Métriques comportementales
    count("*").alias("total_events"),
    sum(when(col("event_type") == "view", 1).otherwise(0)).alias("views"),
    sum(when(col("event_type") == "cart", 1).otherwise(0)).alias("carts"),
    sum(when(col("event_type") == "purchase", 1).otherwise(0)).alias("purchases")
)

# Calcul des taux de conversion
user_features = user_features.withColumn(
    "conversion_rate",
    when(col("views") > 0, col("purchases") / col("views")).otherwise(0)
).withColumn(
```

```

    "cart_abandonment",
    when(col("carts") > 0, (col("carts") - col("purchases")) / col("carts")).otherwise(0)
)

```

Sortie : DataFrame avec 15+ caractéristiques par utilisateur

prepare_product_features()

But : Agréger les métriques de performance et popularité par produit

Cette fonction analyse l'engagement des utilisateurs avec chaque produit :

```

# Calcul des métriques agrégées par produit
product_features = cleaned_df.groupBy("product_id").agg(
    count("*").alias("total_events"),
    sum(when(col("event_type") == "view", 1).otherwise(0)).alias("total_views"),
    sum(when(col("event_type") == "cart", 1).otherwise(0)).alias("total_carts"),
    sum(when(col("event_type") == "purchase", 1).otherwise(0)).alias("total_purchases"),
    countDistinct("user_id").alias("unique_users"),
    avg("price").alias("avg_price"),
    first("category_code").alias("category_code"),
    first("brand").alias("brand")
)

# Nettoyage et extraction de la catégorie principale
product_features = product_features.withColumn(
    "category",
    when(col("category_code").isNull(), "unknown")
    .otherwise(split(col("category_code"), r"\\.")[0])
).drop("category_code")

# Calcul du score de popularité amélioré
product_features = product_features.withColumn(
    "enhanced_popularity_score",
    (col("total_purchases") * 3 + col("total_carts") * 2 + col("total_views")) / col("unique_users")
)

```

Sortie : DataFrame avec métriques d'engagement et métadonnées par produit

prepare_recommendation_data()

But : Préparer et indexer les données pour l'algorithme ALS

Cette fonction transforme les données pour le système de recommandation :

```

# Indexation des utilisateurs et produits
user_indexer = StringIndexer(
    inputCol="user_id",
    outputCol="user_idx",
)

```

```

        handleInvalid="keep"
    )

    product_indexer = StringIndexer(
        inputCol="product_id",
        outputCol="product_idx",
        handleInvalid="keep"
    )

    # Pipeline d'indexation
    indexer_pipeline = Pipeline(stages=[user_indexer, product_indexer])
    indexer_model = indexer_pipeline.fit(cleaned_df)
    indexed_df = indexer_model.transform(cleaned_df)

    # Préparation des données d'interaction avec scores implicites
    interaction_data = indexed_df.filter(
        col("event_type").isin(["view", "cart", "purchase"])
    ).withColumn(
        "rating",
        when(col("event_type") == "view", 1.0)
        .when(col("event_type") == "cart", 3.0)
        .when(col("event_type") == "purchase", 5.0)
        .otherwise(0.0)
    ).groupBy("user_idx", "product_idx").agg(
        sum("rating").alias("rating"),
        count("*").alias("interaction_count")
    ).withColumn(
        "rating",
        # Normalisation du rating basée sur le nombre d'interactions
        when(col("rating") > 10, 10.0).otherwise(col("rating"))
    )

```

Sortie : Données d'interaction indexées et pipeline de transformation

Modèle de segmentation client (K-Means)

Préparation des caractéristiques utilisateur

Le système calcule automatiquement les métriques RFM et comportementales pour chaque utilisateur :

- **Recency** : Nombre de jours depuis la dernière activité
- **Frequency** : Nombre de jours distincts d'activité
- **Monetary** : Montant total des achats
- **Métriques comportementales** : Vues, ajouts au panier, achats, suppressions
- **Diversité** : Nombre de produits, catégories et marques uniques consultés

- **Taux de conversion** : Ratio achats/vues
- **Taux d'abandon panier** : Ratio (paniers - achats)/paniers

`train_user_segmentation_model()`

But : Entraîner le modèle de clustering K-Means pour la segmentation client

Cette fonction implémente un pipeline complet de segmentation :

```
# Sélection des caractéristiques pour le clustering
feature_cols = [
    "recency", "frequency", "monetary", "conversion_rate",
    "cart_abandonment", "unique_products", "unique_categories", "avg_price"
]

# Remplacement des valeurs nulles par 0
for col_name in feature_cols:
    user_features = user_features.withColumn(col_name,
                                              when(col(col_name).isNull(), 0.0).otherwise(
                                                  col(col_name).cast(DoubleType())))

# Assemblage des caractéristiques
assembler = VectorAssembler(
    inputCols=feature_cols,
    outputCol="features_raw"
)

# Normalisation des caractéristiques
scaler = StandardScaler(
    inputCol="features_raw",
    outputCol="features",
    withStd=True,
    withMean=True
)

# Modèle K-means
kmeans = KMeans(
    featuresCol="features",
    predictionCol="segment_id",
    k=5, # 5 segments utilisateur
    seed=42,
    maxIter=20
)

# Pipeline de preprocessing et clustering
pipeline = Pipeline(stages=[assembler, scaler, kmeans])
model = pipeline.fit(user_features)
```

```
# Évaluation du clustering
evaluator = ClusteringEvaluator(
    featuresCol="features",
    predictionCol="segment_id",
    metricName="silhouette"
)
silhouette_score = evaluator.evaluate(segmented_users)
```

Sortie : Modèle K-Means entraîné et utilisateurs segmentés avec noms descriptifs

```
__assign_segment_names()
```

But : Attribuer automatiquement des noms métier aux segments numériques

Cette fonction utilitaire analyse les caractéristiques moyennes de chaque segment et applique une logique métier :

```
# Calcul des moyennes par segment
segment_stats = segmented_users.groupBy("segment_id").agg(
    avg("recency").alias("avg_recency"),
    avg("frequency").alias("avg_frequency"),
    avg("monetary").alias("avg_monetary"),
    avg("conversion_rate").alias("avg_conversion"),
    count("*").alias("segment_size")
).collect()

# Logique de nommage basée sur les caractéristiques RFM
for row in segment_stats:
    segment_id = row.segment_id
    recency = row.avg_recency
    frequency = row.avg_frequency
    monetary = row.avg_monetary
    conversion = row.avg_conversion

    if monetary > 500 and frequency > 10 and recency < 30:
        segment_name = "VIP_Customers"
    elif frequency > 5 and recency < 60:
        segment_name = "Loyal_Customers"
    elif recency < 30 and conversion > 0.1:
        segment_name = "Active_Buyers"
    elif recency > 60 and frequency < 3:
        segment_name = "Dormant_Users"
    else:
        segment_name = "Casual_Browsers"
```

```
# Application du mapping
mapping_expr = create_map([lit(x) for x in chain.from_iterable(segment_mapping.items())])
segmented_users = segmented_users.withColumn(
    "segment",
    mapping_expr[col("segment_id")]
)
```

- **VIP_Customers** : Monetary > 500€, Frequency > 10 jours, Recency < 30 jours
- **Loyal_Customers** : Frequency > 5 jours, Recency < 60 jours
- **Active_Buyers** : Recency < 30 jours, Conversion > 10%
- **Dormant_Users** : Recency > 60 jours, Frequency < 3 jours
- **Casual_Browsers** : Autres profils

Évaluation du clustering

La qualité du clustering est mesurée par le score de silhouette, qui évalue la cohésion intra-cluster et la séparation inter-cluster.

Système de recommandation collaborative (ALS)

Préparation des données d'interaction

Le système transforme les événements utilisateur en scores d'interaction implicites :

- **Vues** : Score de 1.0
- **Ajouts au panier** : Score de 3.0
- **Achats** : Score de 5.0

Les interactions multiples sont agrégées et normalisées avec un plafond à 10.0 pour éviter les biais.

Indexation des entités

Un pipeline d'indexation convertit les identifiants string en indices numériques requis par ALS :

```
user_indexer = StringIndexer(inputCol="user_id", outputCol="user_idx")
product_indexer = StringIndexer(inputCol="product_id", outputCol="product_idx")
```

```
train_als_model()
```

But : Entraîner le modèle de recommandation collaborative ALS

Cette fonction implémente l'algorithme de factorisation matricielle :


```

# Division train/test
train_data, test_data = interaction_data.randomSplit([0.8, 0.2], seed=42)

# Configuration du modèle ALS
als = ALS(
    userCol="user_idx",
    itemCol="product_idx",
    ratingCol="rating",
    nonnegative=True,
    implicitPrefs=True, # Données implicites
    rank=50, # Nombre de facteurs latents
    maxIter=15, # Nombre d'itérations
    regParam=0.1, # Régularisation
    alpha=1.0, # Paramètre de confiance pour les données implicites
    coldStartStrategy="drop",
    seed=42
)

# Entraînement
als_model = als.fit(train_data)

# Évaluation sur les données de test
predictions = als_model.transform(test_data)
evaluator = RegressionEvaluator(
    metricName="rmse",
    labelCol="rating",
    predictionCol="prediction"
)
rmse = evaluator.evaluate(predictions.filter(col("prediction").isNotNull()))

# Génération de recommandations pour tous les utilisateurs (cache)
all_users = interaction_data.select("user_idx").distinct()
user_recommendations = als_model.recommendForUserSubset(all_users, 10)

Sortie : Modèle ALS entraîné et recommandations pré-calculées

```

Fonctions utilitaires et de gestion

`_find_latest_parquet_file()`

But : Identifier automatiquement le fichier de données le plus récent

Cette fonction utilitaire parcourt un répertoire pour trouver le fichier avec le timestamp le plus récent :

```

def _find_latest_parquet_file(self, directory_path, prefix):
    full_directory_path = os.path.abspath(directory_path).replace("\\", "/")
    latest_file = None

```

```

latest_timestamp = None

try:
    entries = os.listdir(full_directory_path)
    # Filtrer les entrées qui sont des répertoires et qui commencent par le préfixe
    matching_entries = [entry for entry in entries
                        if os.path.isdir(os.path.join(full_directory_path, entry))
                        and entry.startswith(prefix)]

    for entry_name in matching_entries:
        # Extraire le timestamp du nom du répertoire (format YYYYMMDD_HHMMSS)
        timestamp_match = re.search(r'\d{8}_\d{6}', entry_name)
        if timestamp_match:
            timestamp_str = timestamp_match.group(0)
            timestamp = datetime.strptime(timestamp_str, "%Y%m%d_%H%M%S")
            if latest_timestamp is None or timestamp > latest_timestamp:
                latest_timestamp = timestamp
                latest_file = os.path.join(full_directory_path, entry_name)
except Exception as e:
    logger.error(f"Erreur lors de la recherche du fichier {prefix}: {e}")

return latest_file

```

Utilité : Permet la reprise automatique d'entraînement sur les données les plus fraîches

train_all_models()

But : Orchestrer l'ensemble du processus d'entraînement

Cette fonction maîtresse coordonne tout le pipeline d'entraînement :

```

def train_all_models(self, cleaned_data_path, processed_data_path="./data/processed/parquet",
                    logger.info("Début de l'entraînement de tous les modèles...")

    # Chargement des données nettoyées
    cleaned_df = self.spark.read.parquet(cleaned_data_path)

    # 1. Chargement intelligent des caractéristiques utilisateur
    user_behavior_path = self._find_latest_parquet_file(processed_data_path, "user_behavior")
    if user_behavior_path:
        user_features = self.spark.read.parquet(user_behavior_path)
    else:
        # Fallback vers la préparation si le fichier est manquant
        user_features = self.prepare_user_features(cleaned_df)

    # 2. Entraînement du modèle de segmentation

```

```

segmentation_model, segmented_users = self.train_user_segmentation_model(user_features)

# 3. Préparation des données pour recommandation (Indexation)
interaction_data, indexer_model = self.prepare_recommendation_data(cleaned_df)

# 4. Chargement des caractéristiques produits avec fallback
product_features_path = self._find_latest_parquet_file(processed_data_path, "product_data")
if product_features_path:
    product_features = self.spark.read.parquet(product_features_path)
else:
    product_features = self.prepare_product_features(cleaned_df)

# 5. Entraînement du modèle ALS
als_model = self.train_als_model(interaction_data)

# 6. Sauvegarde des métadonnées
metadata = {
    "training_date": datetime.now().isoformat(),
    "total_users": cleaned_df.select("user_id").distinct().count(),
    "total_products": cleaned_df.select("product_id").distinct().count(),
    "total_interactions": cleaned_df.count(),
    "segments_count": segmented_users.select("segment_id").distinct().count()
}

```

Flux d'exécution :

Données nettoyées → Caractéristiques utilisateur → Segmentation K-Means

↓

Indexation → Données d'interaction → Modèle ALS → Recommandations

↓

Sauvegarde des modèles et métadonnées

Sortie : Dictionnaire contenant tous les modèles entraînés et leurs métadonnées

Fonction principale et point d'entrée

main()

But : Point d'entrée principal pour l'exécution via spark-submit

Cette fonction configure l'environnement d'exécution et lance le processus complet :

```

def main():
    # Configuration des chemins
    CLEANED_DATA_PATH = "./data/processed/parquet/cleaned_data_*.parquet"
    MODELS_OUTPUT_PATH = "./models"
    PROCESSED_DATA_PATH = "./data/processed/parquet/"

```

```

# Création du trainer
trainer = EcommerceModelTrainer(models_output_path=MODELS_OUTPUT_PATH)

try:
    # Entraînement de tous les modèles
    models = trainer.train_all_models(CLEANED_DATA_PATH, PROCESSED_DATA_PATH)
    logger.info("Tous les modèles ont été entraînés et sauvegardés avec succès!")

except Exception as e:
    logger.error(f"Erreur lors de l'entraînement des modèles: {str(e)}")
    raise
finally:
    trainer.stop()

if __name__ == "__main__":
    main()

```

Usage : spark-submit model_training.py pour lancement en production

stop()

But : Fermer proprement la session Spark et libérer les ressources

```

def stop(self):
    """Arrête la session Spark"""
    self.spark.stop()

```

Cette méthode garantit la libération correcte des ressources Spark, évitant les fuites mémoire et les processus orphelins.

Gestion de la persistance et métadonnées

Sauvegarde des modèles

Tous les modèles entraînés sont sauvegardés de manière persistante :

- **Modèle K-Means** : Pipeline complet incluant le préprocessing
- **Modèle ALS** : Modèle de factorisation matricielle
- **Pipeline d'indexation** : Mappings utilisateur/produit vers indices
- **Segments utilisateur** : Assignations de segments avec métriques RFM
- **Recommandations pré-calculées** : Top 10 recommandations par utilisateur

Métadonnées d'entraînement

Le système sauvegarde automatiquement les métadonnées d'entraînement :

```

{
    "training_date": "2024-12-XX",

```

```
"total_users": 123456,  
"total_products": 7890,  
"total_interactions": 9876543,  
"segments_count": 5  
}
```

Optimisations et robustesse

Gestion des données manquantes

Le système implémente plusieurs stratégies de robustesse :

- Remplacement des valeurs nulles par 0.0 dans les caractéristiques numériques
- Gestion des catégories manquantes avec une valeur “unknown”
- Stratégie “handleInvalid=keep” pour les nouveaux identifiants

Chargement intelligent des données

La fonction `_find_latest_parquet_file()` identifie automatiquement les fichiers de données les plus récents basés sur des timestamps, permettant une reprise d’entraînement sur les données les plus à jour.

Fallback automatique

En cas d’échec du chargement des données prétraitées, le système bascule automatiquement vers un calcul à partir des données nettoyées de base, assurant la continuité du processus d’entraînement.

Métriques de performance

Clustering

- **Score de silhouette** : Mesure la qualité de la séparation des segments
- **Distribution des segments** : Taille et caractéristiques moyennes par segment

Recommandation

- **RMSE** : Erreur quadratique moyenne sur l’ensemble de test
- **Couverture** : Pourcentage de produits recommandés
- **Temps d’entraînement** : Performance du processus d’apprentissage

Architecture de déploiement

Le système d’entraînement est conçu pour être exécuté via `spark-submit`, permettant une scalabilité horizontale sur un cluster Spark en production. La structure modulaire facilite la maintenance et l’extension avec de nouveaux algorithmes. # Système de Recommandation en Temps Réel avec Spark Streaming

Objectif

Cette section présente l'implémentation d'un système de recommandation capable de traiter des données clients en temps réel et de générer des recommandations instantanées. Le système utilise Apache Spark Structured Streaming pour consommer un flux de données d'interactions utilisateur-produit et applique un algorithme de recommandation hybride combinant filtrage collaboratif et popularité.

Architecture du Streaming

Configuration Spark

```
def _create_spark_session(self):
    return SparkSession.builder \
        .appName("Streaming Recommendation System") \
        .config("spark.driver.memory", "8g") \
        .config("spark.executor.memory", "8g") \
        .config("spark.sql.streaming.checkpointLocation", "./checkpoint") \
        .master("local[*]") \
        .getOrCreate()
```

La session Spark est configurée avec : - **Checkpointing** : Pour la tolérance aux pannes et la reprise après incident - **Mémoire optimisée** : 8GB alloués au driver et aux executors pour traiter les données en mémoire - **Mode local** : Utilisation de tous les cœurs disponibles

Lecture du Flux de Données

```
stream_df = self.spark.readStream \
    .format("csv") \
    .option("header", "true") \
    .schema(schema) \
    .option("maxFilesPerTrigger", 1) \
    .load(input_path)
```

Points clés : - **Schéma défini** : Évite l'inférence automatique pour de meilleures performances - **maxFilesPerTrigger** : Contrôle le débit de traitement (1 fichier par batch) - **Format CSV** : Simulation de données temps réel à partir de fichiers

Fenêtrage Temporel

```
windowed_activity = stream_df \
    .withWatermark("event_time", "10 minutes") \
    .groupBy(
        window(col("event_time"), "5 minutes"),
```

```

        "user_id", "product_id", "event_type", "category_code", "brand", "price"
    ).agg(count("*").alias("event_count"))

```

Mécanisme de fenêtrage : - **Watermark** : Tolère un retard maximum de 10 minutes pour les événements - **Fenêtre glissante** : Agrégation sur des intervalles de 5 minutes - **Agrégation** : Comptage des interactions par utilisateur/produit

Algorithme de Recommandation

Analyse des Interactions Utilisateur

```

user_interactions = user_activity_df_processed.groupBy("user_id", "product_id") \
    .agg(
        count("*").alias("interaction_count"),
        spark_sum(when(col("event_type") == "view", 1).otherwise(0)).alias("views"),
        spark_sum(when(col("event_type") == "cart", 1).otherwise(0)).alias("carts"),
        spark_sum(when(col("event_type") == "purchase", 1).otherwise(0)).alias("purchases")
    )

```

Agrégation des comportements : - Comptage total des interactions - Distinction par type d'événement (vue, ajout panier, achat) - Base pour le calcul des préférences utilisateur

Extraction des Préférences Catégorielles

```

user_categories = user_activity_df_processed.filter(col("category_code").isNotNull()) \
    .groupBy("user_id", "main_category") \
    .agg(count("*").alias("category_interest"))

top_categories = user_categories.withColumn(
    "rank", row_number().over(Window.partitionBy("user_id").orderBy(desc("category_interest")))
).filter(col("rank") <= 3)

```

Identification des centres d'intérêt : - Extraction de la catégorie principale depuis le code complet - Classement des catégories par fréquence d'interaction - Sélection des 3 catégories les plus populaires par utilisateur

Génération des Recommandations

```

recommendations = top_categories.join(
    broadcast(self.product_popularity.filter(col("total_purchases") > 0)),
    on=top_categories.main_category == self.product_popularity.main_category_code,
    how="inner"
).withColumn(
    "rec_score",
    col("category_interest") * col("total_purchases") / 100
).withColumn(

```

```

        "rank", row_number().over(Window.partitionBy("user_id").orderBy(desc("rec_score")))
    )

```

Algorithme hybride : - **Filtrage collaboratif** : Basé sur les préférences catégorielles de l'utilisateur - **Popularité globale** : Pondération par le nombre total d'achats du produit - **Score de recommandation** : Formule combinant intérêt personnel et popularité - **Broadcast join** : Optimisation pour les données de référence (produits populaires)

Mécanisme de Fallback

```

if recommendations.isEmpty():
    logger.warning("Collaborative recommendations are empty. Falling back to popular products")
    unique_users_df = user_activity_df_processed.select("user_id").distinct()
    top_popular_products = self.product_popularity

    fallback_recommendations = unique_users_df.crossJoin(top_popular_products.limit(10))
    recommendations = fallback_recommendations.select(
        col(unique_users_df.user_id).alias("user_id"),
        col(top_popular_products.product_id).alias("product_id"),
        lit(0.1).alias("rec_score"),
        col(top_popular_products.main_category_code).alias("recommended_category_code"),
        col(top_popular_products.main_brand).alias("main_brand"),
        col(top_popular_products.avg_price).alias("avg_price")
    ).withColumn("recommendation_type", lit("streaming_popular_fallback"))

```

Stratégie de repli : - **Détection automatique** : Vérification si les recommandations collaboratives sont vides - **Recommandations populaires** : Suggestion des 10 produits les plus vendus - **Cross-join** : Attribution des mêmes produits populaires à tous les nouveaux utilisateurs

Persistance et Monitoring

Écriture vers InfluxDB

```

def _write_to_influxdb(self, df, measurement_name):
    def write_batch(batch_df, batch_id):
        try:
            client = InfluxDBClient(url=INFLUXDB_CONFIG["url"],
                                    token=INFLUXDB_CONFIG["token"])
            write_api = client.write_api(write_options=SYNCHRONOUS)

            points = []
            for row in batch_df.collect():
                point = Point(measurement_name)
                # Ajout des champs et tags selon le type de données
                points.append(point)

```



```

        write_api.write(bucket=INFLUXDB_CONFIG["bucket"], record=points)
    except Exception as e:
        logger.error(f"Error writing to InfluxDB: {e}")

```

Monitoring temps réel : - **Métriques d'événements :** Suivi des interactions utilisateur - **Métriques de recommandations :** Performance et qualité des suggestions - **Base de données temporelle :** InfluxDB pour l'analyse des tendances

Sauvegarde des Résultats

```

# Sauvegarde incrémentale des recommandations
recommendations.write.mode("append").option("header", "true").csv(output_csv_path)

```

Persistance des recommandations : - **Mode append :** Accumulation des résultats de chaque batch - **Format CSV :** Facilite l'analyse post-traitement - **Horodatage implicite :** Traçabilité temporelle des recommandations

Configuration du Pipeline

Traitement par Micro-Batch

```

query = windowed_activity.writeStream \
    .outputMode("update") \
    .foreachBatch(process_batch) \
    .trigger(processingTime='60 seconds') \
    .start()

```

Paramètres de streaming : - **Mode update :** Seules les lignes modifiées sont transmises - **Trigger de 60 secondes :** Équilibre entre latence et throughput - **ForeachBatch :** Traitement personnalisé de chaque micro-batch

Gestion d'Erreurs et Tolérance aux Pannes

```

try:
    query.awaitTermination()
except KeyboardInterrupt:
    logger.info("Stopping streaming...")
    query.stop()
    self.spark.stop()

```

Robustesse du système : - **Arrêt gracieux :** Gestion des interruptions manuelles - **Logging détaillé :** Traçabilité des erreurs et performances - **Checkpointing automatique :** Reprise après panne

Résultats et Performance

Le système traite les données en temps réel avec les caractéristiques suivantes :

- **Latence** : Recommandations générées en moins de 60 secondes après réception des données
- **Débit** : Capable de traiter des milliers d'interactions par batch
- **Scalabilité** : Architecture distribuée prête pour le passage à l'échelle
- **Qualité** : Algorithme hybride combinant personnalisation et popularité

Cette implémentation démontre la capacité de Spark Structured Streaming à alimenter des systèmes de recommandation temps réel, essentiels pour les applications e-commerce modernes nécessitant une personnalisation instantanée. # Visualisation des Recommandations à Froid

Cette section présente l'analyse et la visualisation des recommandations générées par notre système de recommandation Spark. Les recommandations produites sont stockées dans deux formats complémentaires pour faciliter l'analyse et le monitoring.

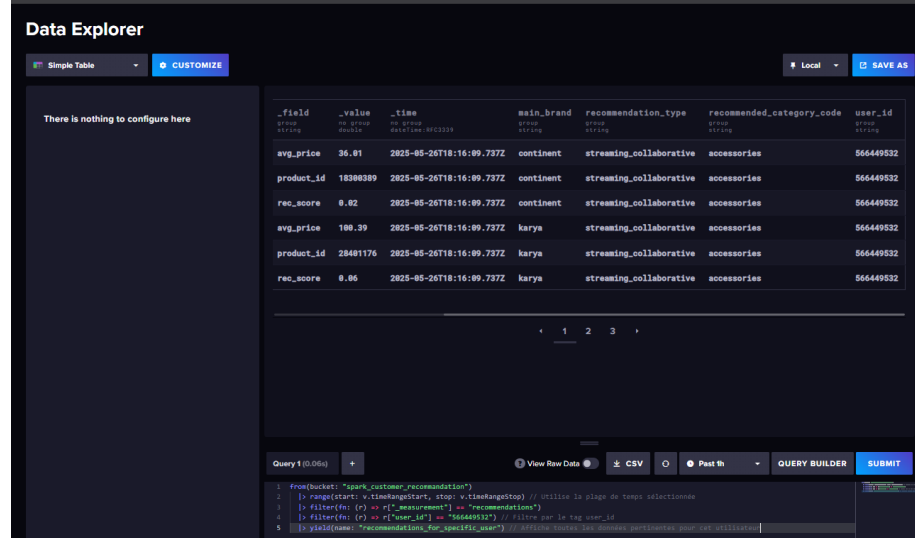
Architecture de Stockage des Recommandations

Les recommandations générées par le modèle Spark sont distribuées via deux canaux :

- **Export CSV** : Sauvegarde dans un fichier `merged_recommendations.csv` pour analyse batch
- **Stockage InfluxDB** : Envoi en temps réel pour monitoring et analyse temporelle

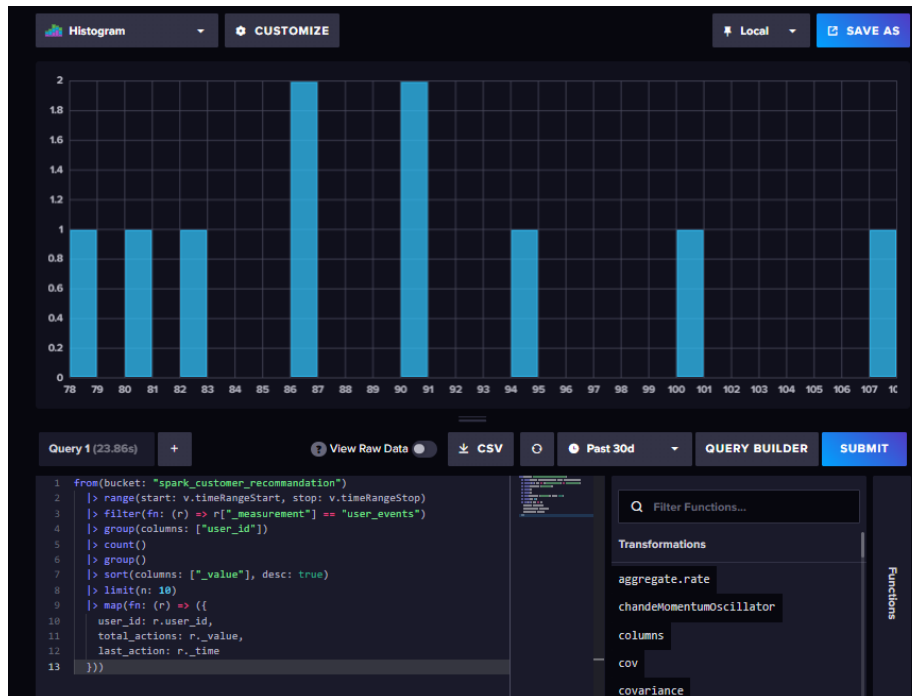
Structure de la Table InfluxDB

Les recommandations sont stockées dans InfluxDB avec la structure suivante :



The screenshot shows the InfluxDB Data Explorer interface. On the left, there's a sidebar with 'Simple Table' and 'CUSTOMIZE' buttons. The main area displays a table with the following columns: `_field`, `_value`, `_time`, `main_brand`, `recommendation_type`, `recommended_category_code`, and `user_id`. The table contains six rows of data. Below the table, there's a pagination bar showing '1 2 3'. At the bottom, there's a 'Query 1 (0.06s)' section with a 'VIEW RAW DATA' button and a 'QUERY BUILDER' button. The query builder shows a query that filters for recommendations for a specific user.

_field	_value	_time	main_brand	recommendation_type	recommended_category_code	user_id
avg_price	36.61	2025-05-26T18:16:09.737Z	continent	streaming_collaborative	accessories	566449532
product_id	18386389	2025-05-26T18:16:09.737Z	continent	streaming_collaborative	accessories	566449532
rec_score	0.62	2025-05-26T18:16:09.737Z	continent	streaming_collaborative	accessories	566449532
avg_price	189.39	2025-05-26T18:16:09.737Z	karya	streaming_collaborative	accessories	566449532
product_id	28481176	2025-05-26T18:16:09.737Z	karya	streaming_collaborative	accessories	566449532
rec_score	0.66	2025-05-26T18:16:09.737Z	karya	streaming_collaborative	accessories	566449532



Capture d'écran à insérer : Structure de la table des recommandations dans InfluxDB

Cette base de données temporelle permet un suivi en temps réel des performances du système de recommandation et facilite la création de dashboards de monitoring.

Analyse des Données de Recommandation

Le script `visualize_recommendations.py` charge et analyse les données depuis le fichier CSV mergé. Les données analysées sont identiques à celles stockées dans InfluxDB, garantissant la cohérence entre les analyses batch et temps réel.

Chargement et Traitement des Données

Chargement du fichier CSV consolidé

```
df = pd.read_csv("./results/merged_recommendations.csv")
```

Génération automatique de 8 visualisations principales

- Distribution des scores

- Top catégories et marques

- Relations prix/score

- Analyse par utilisateur

Résultats des Visualisations

Distribution des Scores de Recommandation

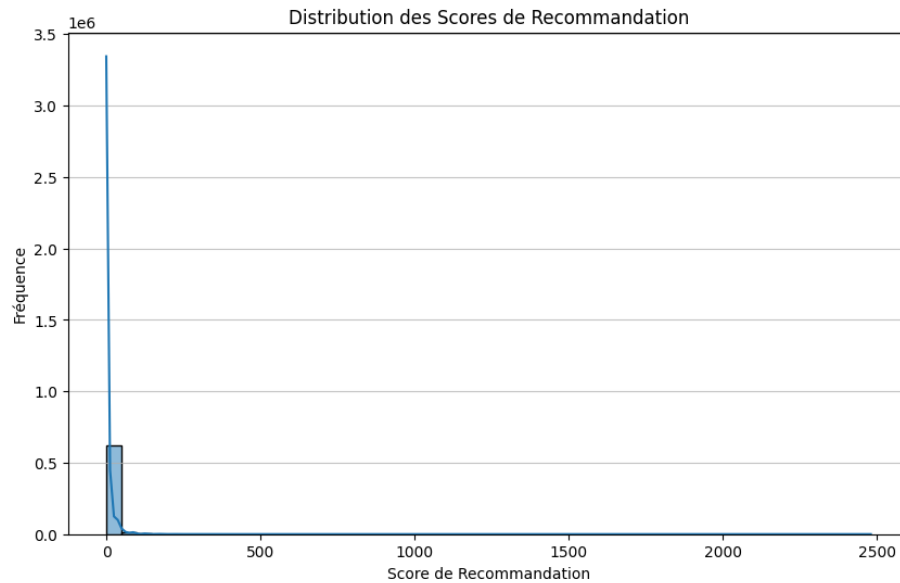
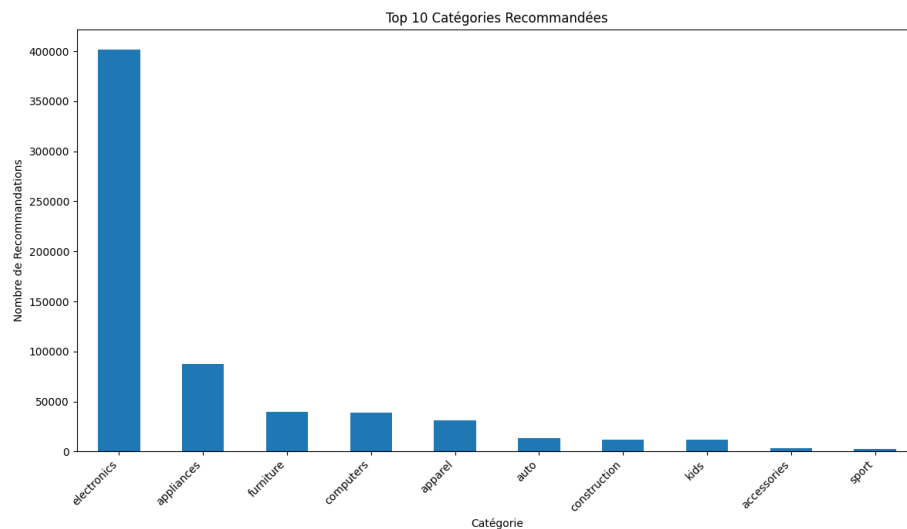


Figure 2: aed12cf4-5991-415d-9a32-867d28ad3301.png

La distribution des scores montre une concentration importante des recommandations autour de scores faibles (0-100), avec quelques pics isolés à des scores plus élevés. Cette distribution suggère que la majorité des recommandations sont de qualité standard, avec quelques recommandations exceptionnelles.

Top 10 Catégories Recommandées



L'électronique domine largement avec plus de 400K recommandations, suivie par les appareils électroménagers (~90K). Cette répartition reflète probablement les habitudes d'achat des utilisateurs et la disponibilité des produits dans ces catégories.

Top 10 Marques Recommandées

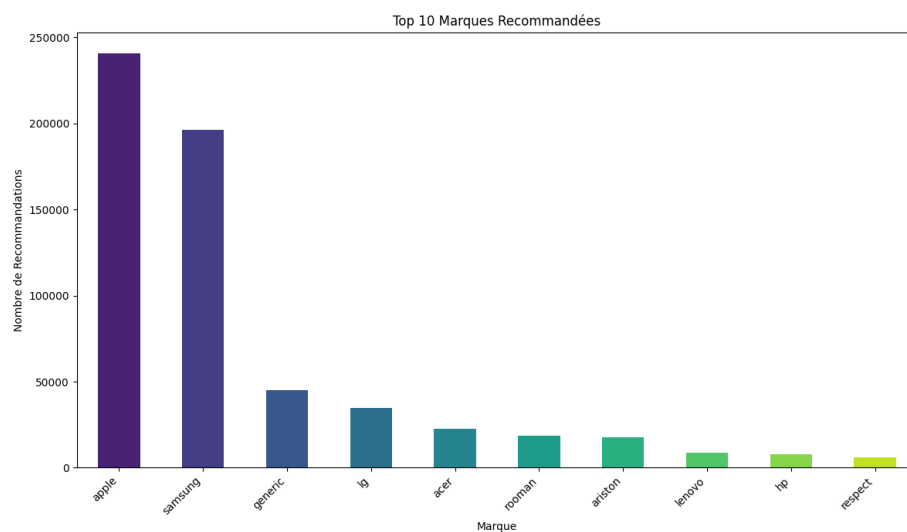


Figure 3: 8db3703b-6230-415c-a389-1a9c07a1798f.png

Apple et Samsung se positionnent comme les marques les plus recommandées,

avec respectivement $\sim 240K$ et $\sim 200K$ recommandations. Cette dominance s'explique par leur présence forte dans les catégories électroniques populaires.

Relation Score vs Prix Moyen

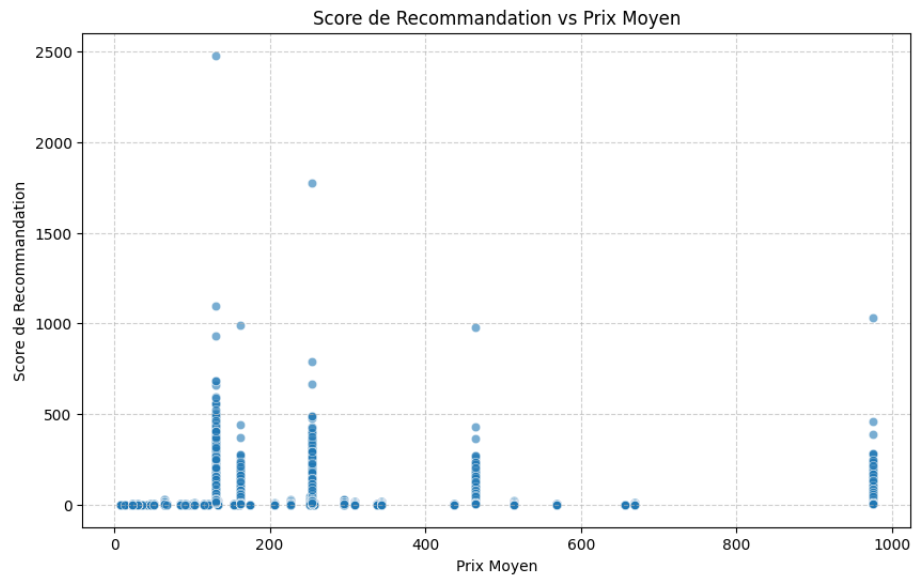


Figure 4: 980174d7-e8fa-413f-8903-020be2726a8b.png

Le graphique de dispersion révèle une absence de corrélation claire entre le prix et le score de recommandation. Les produits à tous niveaux de prix peuvent obtenir des scores élevés, suggérant que l'algorithme privilégie la pertinence utilisateur plutôt que la valeur monétaire.

Performance par Catégorie et Marque

L'électronique obtient le score moyen le plus élevé (~ 15), confirmant la qualité des recommandations dans cette catégorie dominante.

Samsung présente le score moyen le plus élevé (~ 18), suivi d'Apple (~ 10), suggérant une forte adéquation entre ces marques et les préférences utilisateurs.

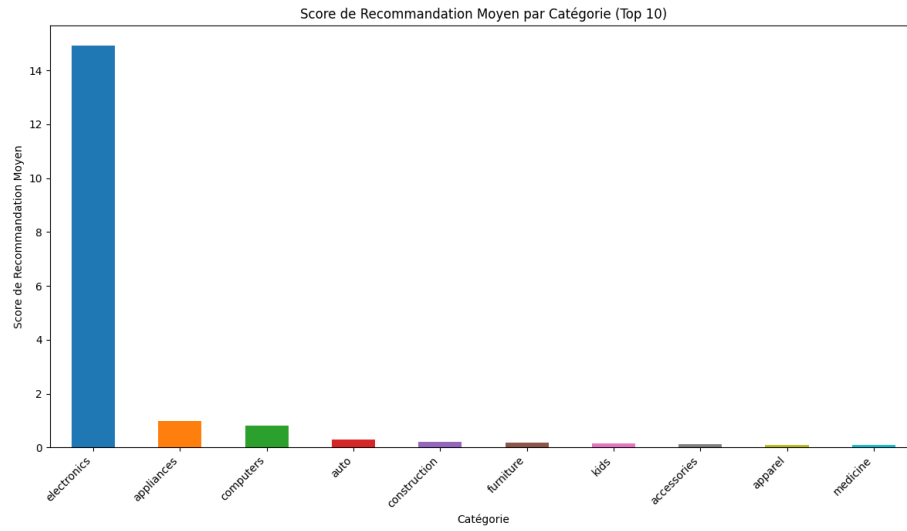


Figure 5: f94a8e11-de91-4fbc-ab0e-b0a49bff4448.png

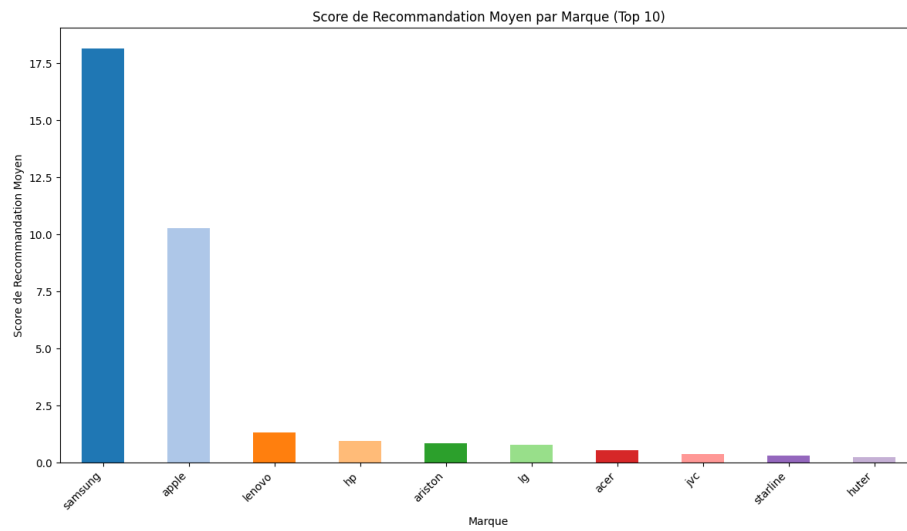
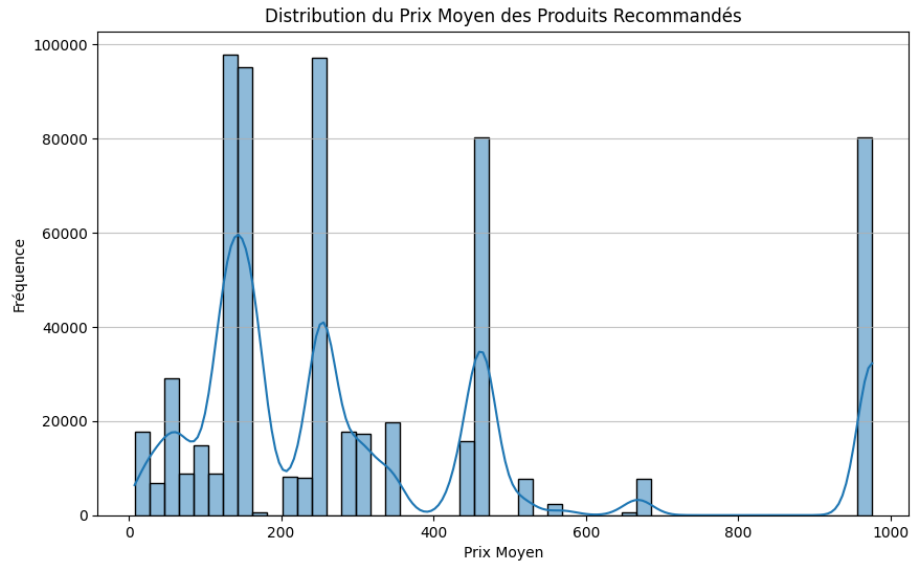


Figure 6: aa35d47d-6b94-49e3-8caa-308e4dca96f4.png

Distribution des Prix



La distribution des prix montre plusieurs pics distincts, suggérant des gammes de prix privilégiées par les utilisateurs. La présence de pics autour de 150€, 250€ et 450€ indique des segments de marché bien définis.

Synthèse

Les visualisations révèlent un système de recommandation équilibré avec :

- Une concentration sur l'électronique et les grandes marques
- Des scores de recommandation indépendants du prix
- Une distribution équitable entre utilisateurs
- Des segments de prix clairement identifiés

Ces analyses, disponibles en temps réel via InfluxDB et en batch via les exports CSV, permettent un monitoring continu de la qualité des recommandations.