TP FD: Apriori optimization

Belloula Ahmed amine	Daddi hammou mahdi
202035014988	202039080304
G1 M1-MV	G1 M1-MV

Dataset

Nous avons appliqué l'algorithme apriori sur l'ensemble de données Market Basket, où chaque ligne représente une transaction et chaque colonne représente un élément différent. Cet ensemble de données comprend 7 501 transactions et 119 éléments distincts.

Code apriori

```
def apriori(df: pd.DataFrame):

    frequent_item_sets = defaultdict(list)

    print("level 0: ", end=" ")
    means = df.mean()
    min_support = means.mean()
    # first level candidates are the items themselves, filter them by min_support
    for i, support in enumerate(means[means >= min_support]):
        frequent_item_sets[0].append(({i}, support))

    print(f"{len(frequent_item_sets[0])} frequent items.")

    total_itemsets = 0
    for level in range(1, len(df.columns)):
        print(f"level {level}: ", end=" ")

        current_level_candidates = generate_candidate_item_sets(
            frequent_item_sets, level
        )

        if len(current_level_candidates) == 0:
            print(f"{level} reached, no more frequent item sets.")
            break
        else:
```

```
            print(f"{len(current_level_candidates)} candidates.", end=" ")

        frequent_item_sets[level] = prune_candidates(df, current_level_candidates)

        total_itemsets += len(frequent_item_sets[level])

        print(
            f"{len(frequent_item_sets[level])} frequent item sets, min support:
            {min_support}"
        )

    print(f"Total frequent item sets: {total_itemsets}")
    return frequent_item_sets
```

Règles d'association

```
def association_rules(min_confidence, min_lift, support_dict) -> pd.DataFrame:
    rules_data = []

    for itemset, support in support_dict.items():

        if len(itemset) < 2:
            continue

        for i in range(1, len(itemset)):
            for A in combinations(itemset, i):
                A = frozenset(A)
                B = itemset - A

                support_A = support_dict[A]
                support_B = support_dict[B]

                confidence = support / support_A
                lift = support / (support_A * support_B)

                if confidence >= min_confidence and lift >= min_lift:
                    rules_data.append(
                        {"A": A, "B": B, "Confidence": confidence, "Lift": lift}
                    )

    rules_df = pd.DataFrame(rules_data)
    return rules_df
```

Préparation des données

Nous avons créé un script personnalisé qui transforme un ensemble de données de transaction en un ensemble binarisé, en utilisant: `pandas.get_dummies()`

cela nous aidera à accélérer les calculs du support minimum, car nous pouvons utiliser des opérations vectorielles efficaces pour compter le nombre de transactions dans lesquelles chaque ensemble d'articles apparaît. Cela réduit la

complexité et accélère considérablement le processus de calcul du support minimum.

```
def main(filename):

    try:
        df = read_file(filename)

        # Binarize the transactions
        binarized_df = (
            pd.get_dummies(df.stack(), prefix="", prefix_sep="").groupby(level=0).max()
        )

        # Convert True and False to 1 and 0
        binarized_df = binarized_df.replace({True: 1, False: 0})

        # Create a new filename by appending "_binarized" before the file extension
        base_name, extension = os.path.splitext(filename)
        new_name = f"{base_name}_binarized{extension}"

        # Save to CSV file
        binarized_df.to_csv(new_name, index=False, header=True)

        print(f"Binarized file '{new_name}' created successfully!")
    except ValueError as e:
        print(e)
        sys.exit(1)
```

Optimisations

Utilisation de pandas

pandas est un puissant outil d'analyse de données construit sur Numpy, nous avons choisi d'utiliser pandas en raison de son API simple et de ses calculs rapides utilisant la vectorisation.

pour montrer à quelle vitesse il est, nous avons créé un benchmark, pour mesurer la différence entre le python ordinaire et les pandas.

```
def get_support_without_pd(df, itemset):
    if len(df) == 0:
        return 0.0

    supports = []
    for index, row in df.iterrows():
        all_true = True
        for i in sorted(itemset):
            if not row[i]:
                all_true = False
                break
        if all_true:
            supports.append(1)
        else:
```

```
            supports.append(0)

    return sum(supports) / len(df)
```

Nous utilisons la bibliothèque `timeit` pour 100 itérations sur un ensemble d'éléments `{0, 1, 2}`

et nous obtenons ces résultats :

Pandas	Without Pandas
0.06 seconds	26.67 seconds

Une amélioration de 444,5 fois !.

Combinations

Python dispose d'une fonction intégrée pour générer des combinaisons d'éléments uniques, fournie dans la bibliothèque `itertools`. ce qui est plus propre et plus rapide pour générer des candidats de niveau supérieur.

```
def generate_candidate_item_sets(frequent_item_sets, level):
    current_level_candidates = list()

    if len(frequent_item_sets[level - 1]) == 0:
        return current_level_candidates

    # Extract unique items from the frequent item sets of the previous level
    unique_items = set()
    prev_level_sets = list()
    for item_set, _ in frequent_item_sets[level - 1]:
        unique_items.update(item_set)
        prev_level_sets.append(item_set)

    # Generate candidates by combining unique items
    for candidate_set in combinations(unique_items, level + 1):
        # convert the tuple to a set
        candidate_set = set(candidate_set)
        # if the candidate set has a subset that doesn't exist in the
        # frequent item sets of the previous level, skip it
        if is_valid_set(candidate_set, prev_level_sets):
            current_level_candidates.append(candidate_set)

    # calculate the support of each candidate set
    return current_level_candidates
```

Règle de monotonie

Nous exploitons ce concept dans la fonction `is_valid_set()`

```
def is_valid_set(item_set, prev_level_sets):
    """
    Check if all the subsets of the item_set are present in the previous level sets.

    Parameters:
        item_set (list): The item set to be validated.
```

```
        prev_level_sets (list of lists): List of sets from the previous level.

    Returns:
        bool: True if all subsets of the item_set are present in prev_level_sets, False oth
    """
    if len(prev_level_sets) == 0:
        return False

    single_drop_subsets = get_subsets(item_set)
    for single_drop_set in single_drop_subsets:
        if single_drop_set not in prev_level_sets:
            return False

    return True
```

Example:

- `prev_level_sets = [{1, 2}, {2, 3}, {1, 3}]`
- `item_set = {1, 2, 4}`

dans cet exemple, un sous-ensemble de item_set : `{1, 4}` n'est pas présent dans `prev_level_sets`, suivant le principe selon lequel si un ensemble n'est pas fréquent, tous ses sur-ensembles ne sont pas non plus fréquents, et puisque `{1, 4}` n'est pas fréquent, c'est-à-dire : non présent dans `prev_level_sets`, `{1, 2, 4}` n'est pas non plus fréquent.

Min support dynamique

à chaque itération, nous élaguons les éléments en fonction de la moyenne de leur support, cela présente plusieurs avantages :

- Support min dynamique qui s'ajuste par niveau.
- La conservation des informations puisque le support min est majoritairement choisi par le support du premier niveau.

Analyse du vitesses

```
time_taken_apriori = timeit.timeit(lambda: apriori(df), number=num_iterations)

Time taken for 100 iterations of apriori: 46.38617599999998 seconds
```

46 secondes pour analyser 100 fois l'ensemble de données qui contient 7501 × 119 éléments. ce qui signifie 75 010 lignes en 46 secondes, soit environ 1 630 lignes/s

```
time_taken_association_rules = timeit.timeit(
    lambda: association_rules(min_confidence, min_lift, item_support_dict),
    number=num_iterations,
)

Time taken for 100 iterations of association_rules: 0.14365209999959916 seconds
```