Technical Approach

The Dataset

We worked with a dataset that consisted of the following columns:

- Transaction ID
- Item
- Date/Time
- DayPart
- DayType

The dataset contained over 20,000 rows with over 9,000 transactions. The data was initially provided by Kaggle as a csv file. We parsed this data using the pandas python library, converting the csv file into a dataframe. From this point, we could preprocess the data in order to filter out any erroneous or unnecessary entries.

Preprocessing

Given the high number of results we intended to produce with this dataset, it made little sense to preprocess the data the same way for each step of the technical approach. A quick audit of the dataset revealed that it was clean to begin with, requiring little preprocessing for calculating Support, Confidence, and Lift.

In order to run the FP-Growth algorithm, we had to create a dictionary of transactions, each containing all items in that transaction. Given that the original dataset contained purchased items correlated simply with a transaction ID, we had to iterate over the dataset and add each transaction ID to a dictionary, adding any items found in that transaction ID to the dictionary as well. After this was done, we had the dictionary required to run the FP-Growth algorithm.

Support & Confidence

Support is a key metric in identifying which items are most popular within a given dataset. The motivation behind using such a metric was to learn what items should be stocked to what amounts on any given day. Further, finding the support of each item during a given season, day of the week, or part of a day gave further insight into what items should be stocked to what amounts in any specific case. Calculating support was simple; we counted the number of entries matching the criteria we were looking for. We once again used python to calculate both the support and confidence.

Confidence is another key metric in identifying which items in a dataset may be bought together. The motivation behind using such a metric is to find what items can be offered grouped as packages to increase sales. Calculating confidence was similarly simple; we created a dataset consisting of all items that fit the antecedent, then counted the proportion of rows which contained the consequent item.

FP-Growth

We then used FP-Growth to find rule associations in our dataset. In order to run the FP-Growth algorithm, we used the mlxtend python library. However, for the technical approach, a summary of the algorithm is as follows. FP-Growth involved two main parts: the construction of the FP-tree and the mining of frequent itemsets. Constructing a FP-tree involves building an FP-tree consisting of nodes, each of which represents an item. The paths from the root to the leaf nodes represent transactions. The primary goal of this data structure is to compactly store the items in the dataset and their relationships. The algorithm scans the dataset only once to construct the FP-tree. Transaction reduction is then performed by merging transactions with identical itemsets, thereby reducing the memory required for the FP-tree. After the FP-Tree has been created, it is mined for association rules. The tree is then recursively traversed, finding the most associated items using conditional pattern bases.

The motivation behind using FP-Growth, and, more broadly the use of association rules, is to find items that are highly correlated to offer them as sales together. This is similar to the confidence metric; however, this approach is far more targeted.

Lift

Lift measures the predictive power of a rule association generated by FP-Growth by comparing the observed support of the combined items in the rule with the support expected if those items were independent of each other. For our purposes, lift helps in determining the strength and direction of association between items. While rule associations between the most popular items and every other item will be prominently shown as results of the FP-Growth algorithm, these results are not particularly interesting. Since we already know that coffee is the most popular item at a cafe, it is important to find items that have strong correlations, even if they are less popular themselves, in order to find the best targeted advertisements and deals.

Naive Bayes

Another aspect of our dataset that we wished to reveal was which items are most frequently purchased together during each season. Our first attempt at uncovering this was to train a Naive bayes classifier on our data. Though we used the scikit-learn MultinomialNB classifier to make our predictions, for the purpose of explaining our technical approach, we will briefly describe how a Multinomial Naive Bayes classifier works. Bayes classifiers are a probabilistic framework for classification based on Thomas Baye's logical theorem commonly known as Baye's law. Naive Bayes classifiers, as opposed to Bayesian networks, assume that features are all conditionally independent of one another given the classification label they are assigning. This assumption is the source of the Naive prefix in Naive Bayes as, while it may allow for more efficient modeling, its lack of accounting for conditional dependence can cause the model to miss connections between features and give less accurate results when used on correlated attributes. Despite this shortcoming, Naive Bayes classifiers still tend to perform well at predicting classification of items, but more specifically, Multinomial Naive Bayes classifiers tend to excel at classifying text when using discrete features or features that can be easily encoded into integers.

The MultinomialNB assumes that its features follow a multinomial distribution, which is then used alongside a manipulated equation from Bayes law to estimate the probability of observing specific features for each possible classification of its target variable, which, in our case, was the season in which the transaction took place. The scikit-learn MultinomialNB also uses Laplace smoothing to handle features that aren't seen in a sample of training data, and finally uses logarithmic probabilities in the last step of the prediction to avoid underflow errors due to the multiplication of many very small probabilities. In the end, due to issues with the distribution of our data and the fact that items purchased together are likely to be correlated, we were unable to bring the accuracy score of our MultinomialNB Classifier to a level that we found sufficient, so we decided to use a different method of classification.

Decision Tree Classifiers

Our next attempt at predicting the season of transactions to estimate which items will sell together most frequently was to use a decision tree classifier. For this method, we again used a scikit-learn class, DecisionTreeClassifier. A decision tree is a model that simulates decisions and the possible outcomes in which they result. Inside a decision tree model, internal nodes represent tests on an attribute, branches between nodes represent an outcome of said tests, and leaf nodes represent class labels or classifications. Because the number of decision trees that can be built from a set of attributes rises exponentially as the attribute count increases, one must use 'greedy strategies' to prioritize attribute tests that maximize information gain and/or minimize node impurity. Scikit-Learn's DecisionTreeClassifier uses an optimized version of the CART tree building algorithm, which selects the next attribute to be tested at each level based on which attribute will result in the lowest amount of entropy or gini impurity. Branches that do not significantly improve performance are then pruned, and the tree-building continues. Once the tree is built, test data can be passed into it and classified based on its feature values.

We found much better results with the decision tree classifier, which we used by first encoding our categorical data of items purchased in a transaction into an array of binary values:

```
# one-hot encode the non-normalized items
te = TransactionEncoder()
te_ary = te.fit(transactions_with_season['Items']).transform(transactions_with_season['Items'])
encoded_items = pd.DataFrame(te_ary, columns=te.columns_)
```

We then pulled the season of each transaction based on the date it took place and split this data at an 80:20 ratio of training to test data:

```
# split data into training/testing sets
X_train, X_test, y_train, y_test = train_test_split(encoded_items, transactions_with_season['Season'], test_size=0.2, random_state=250)
```

Next we used recursive feature elimination to find the most important attributes for

classification and fit our decision tree model to our training data:

```
# select features with RFE
tree = DecisionTreeClassifier()
rfe = RFE(estimator=tree, n_features_to_select=90)
rfe.fit(X_train, y_train)
selected_features = X_train.columns[rfe.support_]

# train decision tree with selected features
tree_selected_features = DecisionTreeClassifier()
tree_selected_features.fit(X_train[selected_features], y_train)
```

Then, using this model, we predicted the season for each transaction in our test data, achieving more than double the accuracy we had with the Naive Bayes classifier.

```
# predict season for each itemset in test data
predicted_seasons = tree_selected_features.predict(X_test[selected_features])
```

From there, we took these predicted seasons and recombined them with the test data to predict item combinations with the highest probability of being purchased together for each season.

```
# combine predictions with test data
combined_data = pd.DataFrame({
    'Items': X_test.apply(lambda row: ','.join(X_test.columns[row == 1]), axis=1),
    'Season': y_test.values,
    'Predicted_Season': predicted_seasons
})
```

```
# split items column into strings and get the number of items in each transaction
combined_data['Num_Items'] = combined_data['Items'].str.split(',').apply(len)

# filter out itemsets with only one item
combined_data_filtered = combined_data[combined_data['Num_Items'] > 1]

# drop 'Num_Items' column after filtering
combined_data_filtered.drop(columns=['Num_Items'], inplace=True)

# make dataframe from predicted seasons
predicted_seasons_df = pd.DataFrame(predicted_seasons, index=combined_data.index, columns=['Predicted_Season'])

# filter predicted seasons to match the filtered test data
predicted_seasons_filtered = predicted_seasons_df.loc[combined_data_filtered.index]

# combine predicted seasons with test data
combined_data_filtered['Predicted_Season'] = predicted_seasons_filtered
```

```
# iterate over predicted seasons
for predicted_season in [1,2,3,4]:
    # filter combined data to the current season
    seasonal_data = combined_data_filtered[combined_data_filtered['Predicted_Season'] == predicted_season]

    # store counts of itemsets for current predicted season in a dictionary
    itemset_counts = {}

    # iterate over each itemset in filtered data
    for itemset in seasonal_data['Items']:
        # split itemset string into list of items
        items_list = itemset.split(',')

        # iterate over each combination of items in itemset
        for r in range(2, len(items_list) + 1):  # start combinations at 2 items
            for combination in combinations(items_list, r):
                # convert combination to tuple
                combination_tuple = tuple(combination)

                # increase count for combination in itemset counts
                if combination_tuple not in itemset_counts:
                    itemset_counts[combination_tuple] = 0
                itemset_counts[combination_tuple] += 1

    # sort itemset counts dictionary by count
    sorted_itemset_counts = {k: v for k, v in sorted(itemset_counts.items(), key=lambda item: item[1], reverse=True)}

    # store sorted itemset counts for current predicted season
    predicted_season_itemset_counts[predicted_season] = sorted_itemset_counts

top10_predicted_itemsets = []
# print top 10 itemsets for each predicted season
for predicted_season, itemset_counts in predicted_season_itemset_counts.items():
    #print(f"Predicted Season: {predicted_season}")
    counter = 0
    for itemset, count in itemset_counts.items():
        top10_predicted_itemsets.append((itemset,predicted_season, count))
        #print(f"Itemset: {itemset}, Count: {count}")
        counter += 1
        if counter == 10:
            break
```

After doing all of this, we still had a relatively low accuracy score:

```
accuracy = tree_selected_features.score(X_test[selected_features], y_test)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.4585314315900686 5
```

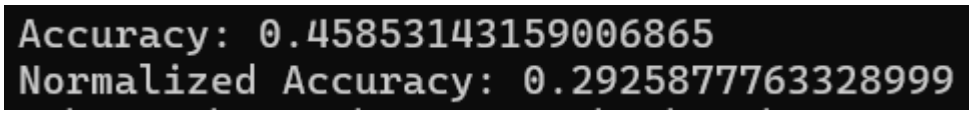
We believed this low score may have come from the uneven distribution of our data, as ~77% of the data was for the seasons of winter and spring, leaving only 23% of the data for the entirety of summer and fall. To see if this imbalance was the problem, we attempted to train a decision tree on normalized data with the same amount of transactions for each season:

```
# find the season with the minimum number of transactions
min_transactions = transactions_with_season['Season'].value_counts().min()

# make a new dataframe to store season-normalized data
normalized_transactions_with_season = pd.DataFrame(columns=transactions_with_season.columns)

# get equal amount of transactions for each season
for season in [1, 2, 3, 4]:
    season_data = transactions_with_season[transactions_with_season['Season'] == season]
    normalized_season_data = season_data.sample(min_transactions, random_state=250)
    normalized_transactions_with_season = pd.concat([normalized_transactions_with_season, normalized_season_data])
```

But unfortunately, even with many adjustments to the hyperparameters of our decision tree, we consistently got significantly lower accuracy from the normalized data:



This left us with the conclusion that we simply did not have enough data to classify summer and fall transactions using items as features.