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 litany of results we would like to produce with this dataset, it made little sense to preprocess the data the same way for each step of the technical approach. For calculating Support, Confidence, and lift little preprocessing was needed. A quick audit of the dataset revealed that it was very clean initially, requiring little preprocessing.

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 simply had items bought correlated 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 is done, we have the dictionary required to run the FP-Growth algorithm.

Support & Confidence

Support is a key metric in finding what items are most popular in a given dataset. The motivation behind using such a metric was to find 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, gives further insight into what items should be stocked to what amounts in a given specific case. Calculating support is simple, just count the number of entries that match the criteria you are looking for. We once again used python to calculate both the support and confidence.

Confidence is another key metric in finding what items may be bought together in a dataset. The motivation behind using such a metric is to find what items can be offered as packages together to increase sales. Calculating confidence is similarly simple to calculating support, simply create a dataset consisting of all items that fit the antecedent. Then count the proportion of rows which contain 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 the following. FP-Growth involves two main parts: the construction of the FP-tree and the mining of frequent itemsets. Constructing the 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, which reduces the memory required for the FP-tree. After creating the FP-Tree, the FP-tree 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 much 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 expected support 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. We used the scikit-learn MultinomialNB classifier to make our predictions, but for the purpose of explaining our technical approach we will briefly describe how a Multinomial Naive Bayes classifier works. Bayes classifiers are 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, the lack of accounting for conditional dependence can cause the model to miss connections between different features and give less accurate results when being used on correlated attributes. Despite this, 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 the transaction took place in. 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 due to the fact that items purchased together are likely to be correlated, we were unable to get the accuracy score of our Naive Bayes Classifier up to a level that we could work with, 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 they result in. 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 make prioritize attribute tests that that maximize information gain 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 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, then pulling the season of each transaction from the date it took place and splitting this data at an 80:20 ratio of training data to test data. We then used recursive feature elimination to find the most important attributes for classification and fit our decision tree model to our training data. Then using this model we predicted the season of each transaction in our test data, getting more than double the accuracy we found with the Naive Bayes classifier. From there, we took these predicted seasons and recombined them with the test data to predict the item combinations with the highest probability of being purchased together for each season.