Report

March 24, 2019

1 Project #1 (Mini-project): Frequent Itemset Mining

Erin Witmer CSC 440

1.1 Objective

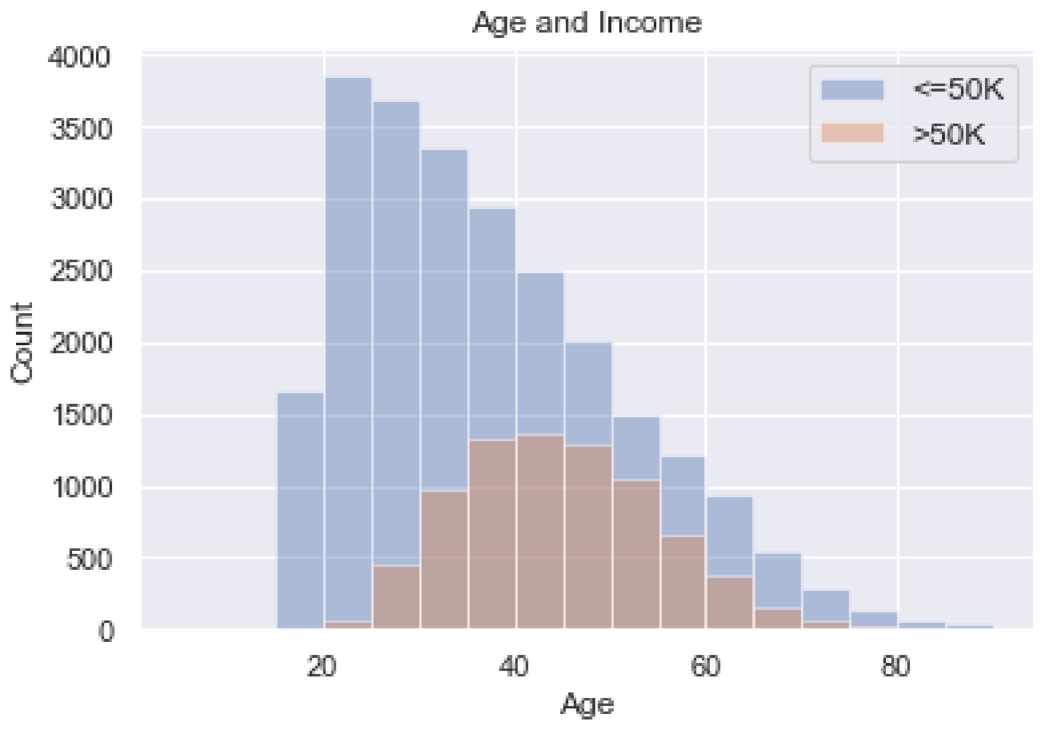
The assignment is to mine the UCI Adult Census Dataset for frequent itemsets using the Apriori algorithm (with one improvement) and the FPGrowth algorithm. Both algorithms were written without the assistance of any frameworks or packages. An enhancement to the Apriori algorithm using partitioning was also written.

1.1.1 Step 1: Data Visualization and Cleaning

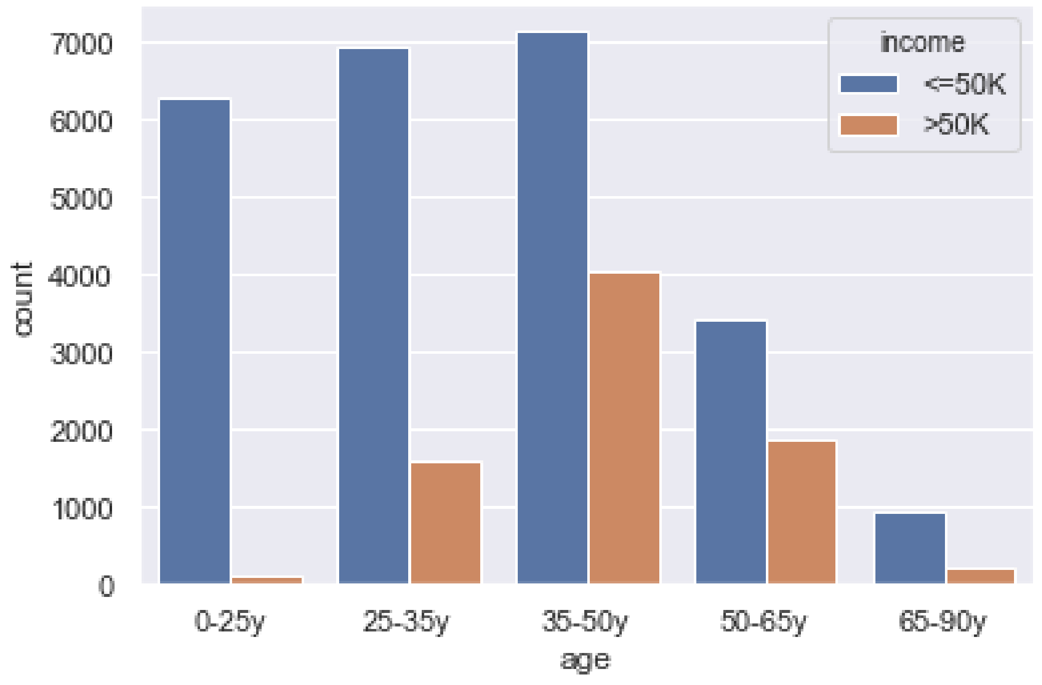
The dataset is in raw format. Effective analysis starts with understanding the dataset basic characteristics, analyzing each feature and cleaning the dataset. The basic process for each feature is: - visualize the data - perform any binning/modifications to the feature set - update the dataset with the clean data

Age First, split the data between transactions where income is <=50K and >50K. Then bin the data by 10 years to visualize where the optimal breakpoints are for final binning of the data.

```
Out[259]: Text(0,0.5,'Count')
```

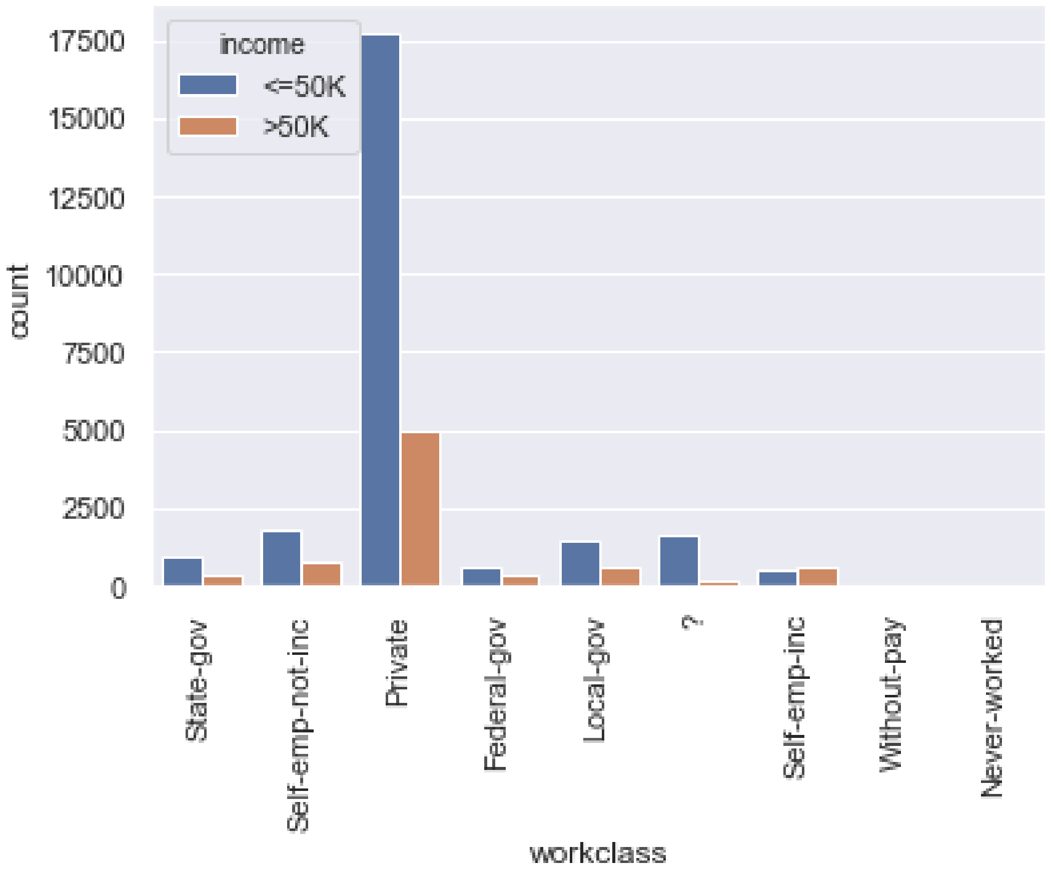


The following histogram shows the bins that were used in the analysis:



Workclass No modifications were made to the data.

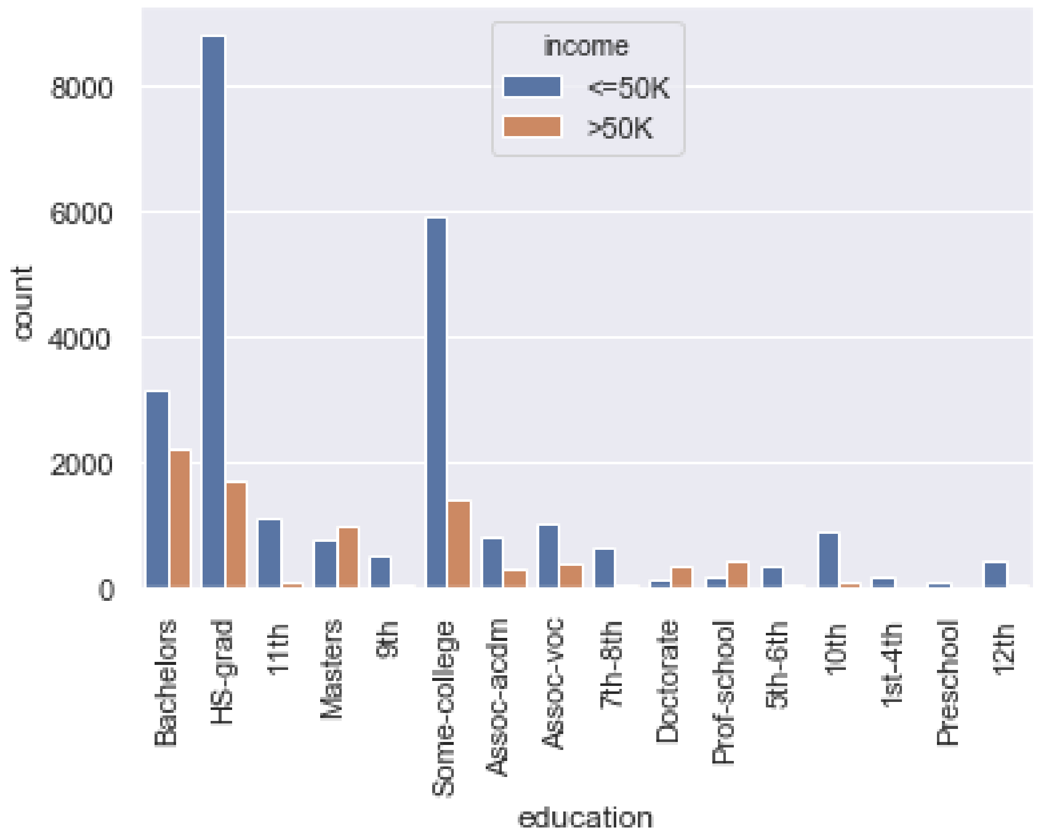
```
Out[263]: (array([0, 1, 2, 3, 4, 5, 6, 7, 8]), <a list of 9 Text xticklabel objects>)
```



Final Weight This feature was dropped, we can assume the transactions in the dataset represent a sample of the population.

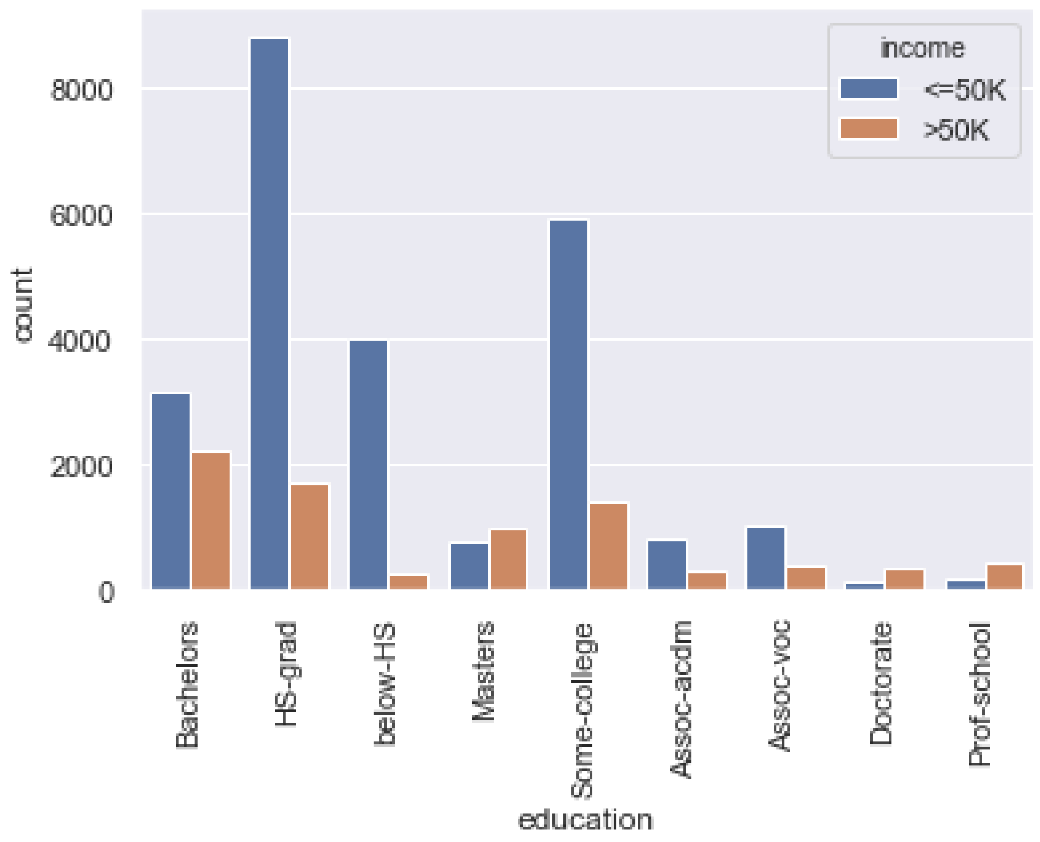
Education Visualizing the data, there appear to be a number of categories that can be combined.

```
Out[265]: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15]),
           <a list of 16 Text xticklabel objects>)
```



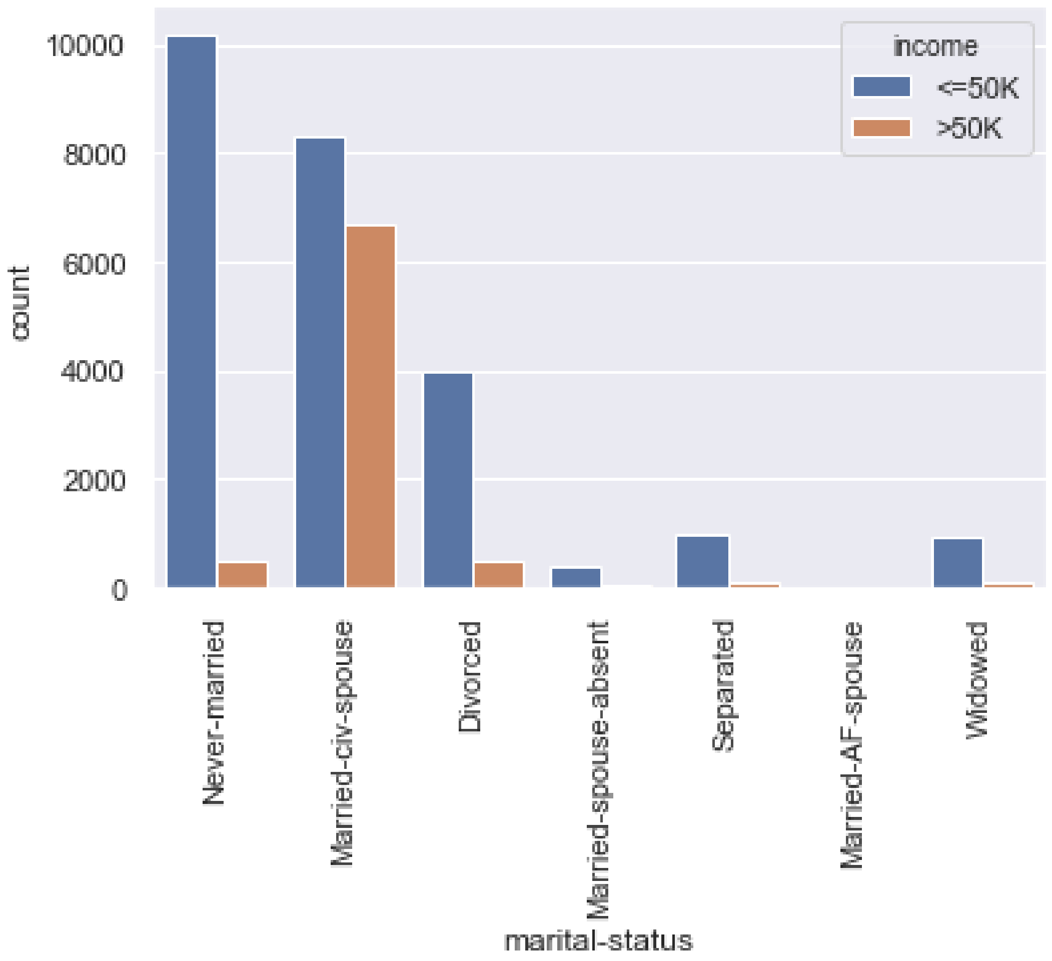
All categories that represent an education level below high school graduate are combined into one category. Since the continuous variable representing education level is repetitive, it is deleted from the dataset.

```
Out[266]: (array([0, 1, 2, 3, 4, 5, 6, 7, 8]), <a list of 9 Text xticklabel objects>)
```



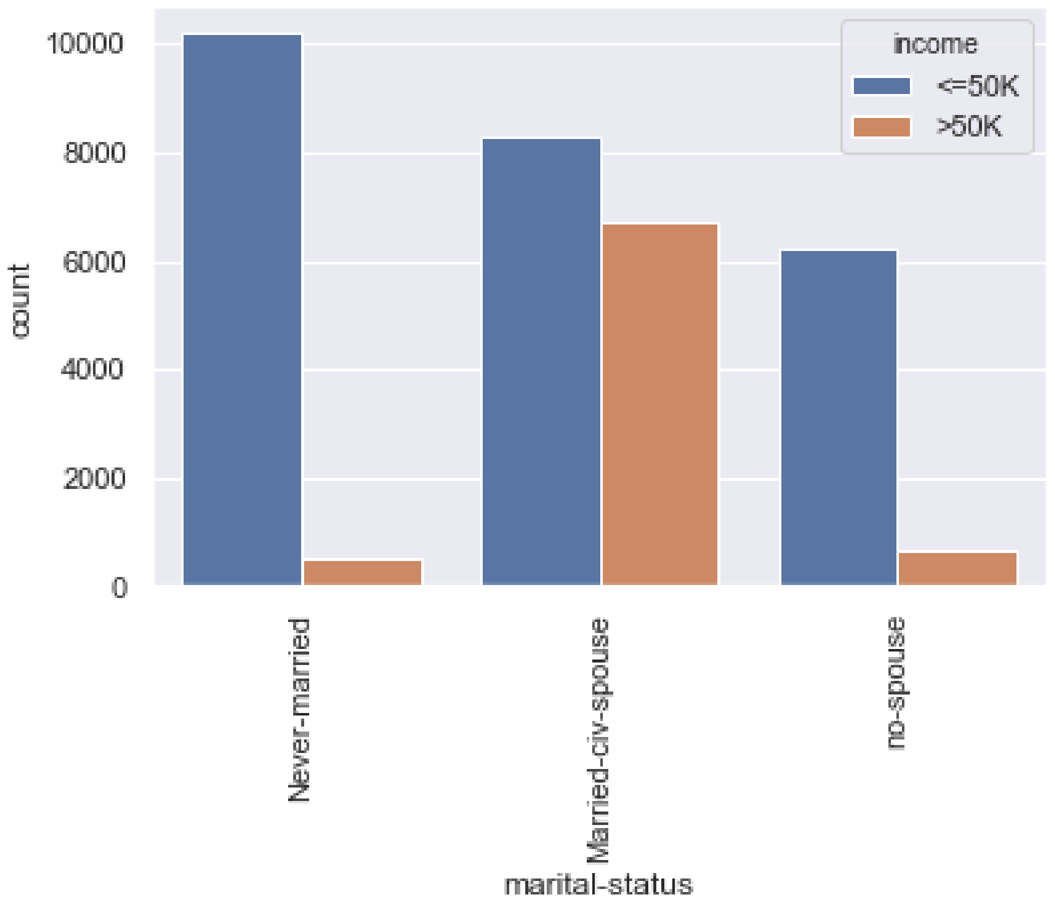
Marital Status Visualizing the marital status data, a number of categories can be combined to simplify the data.

```
Out[267]: (array([0, 1, 2, 3, 4, 5, 6]), <a list of 7 Text xticklabel objects>)
```



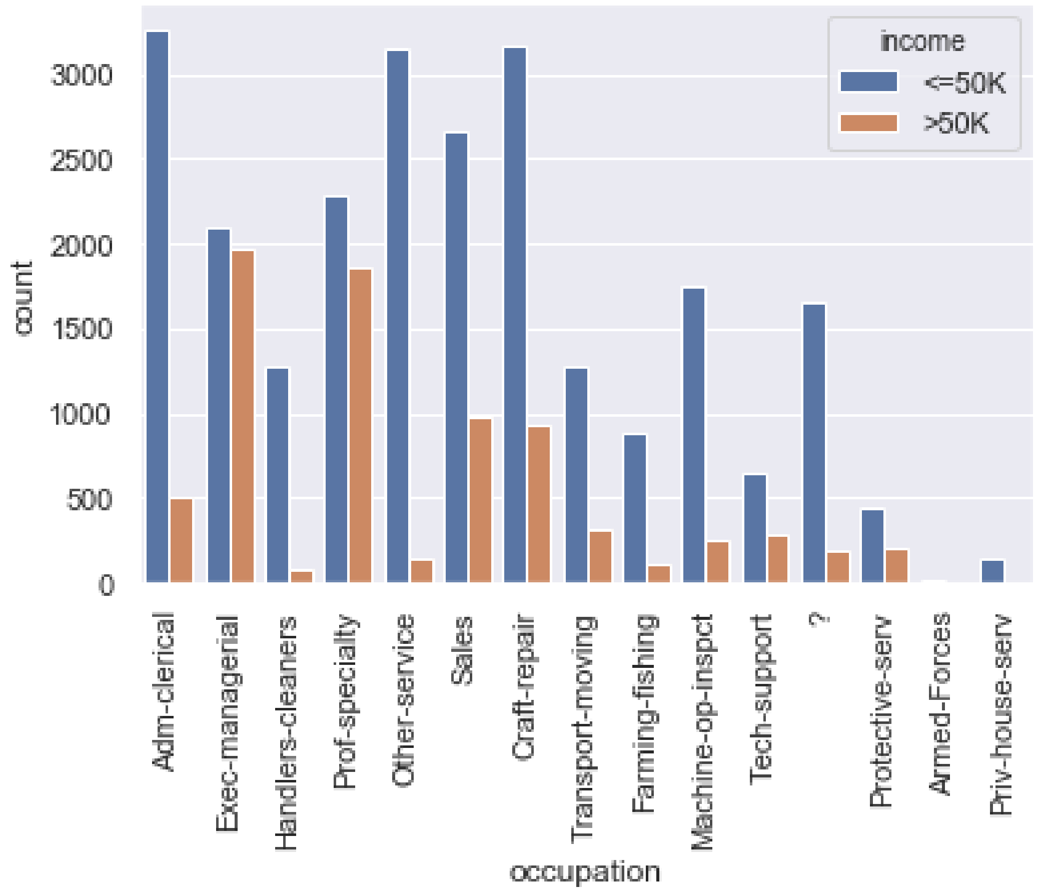
The final categories used in the analysis are shown below.

```
Out[268]: (array([0, 1, 2]), <a list of 3 Text xticklabel objects>)
```



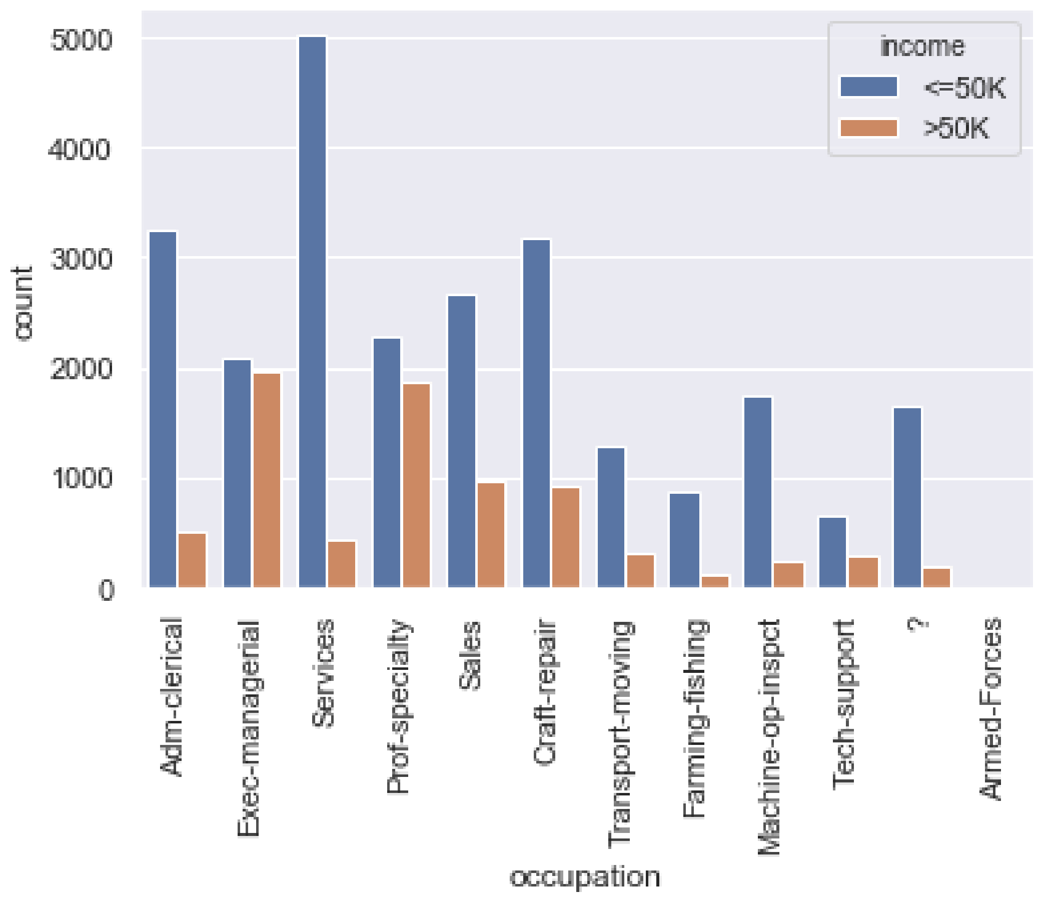
Occupation

```
Out[269]: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14]),
           <a list of 15 Text xticklabel objects>)
```



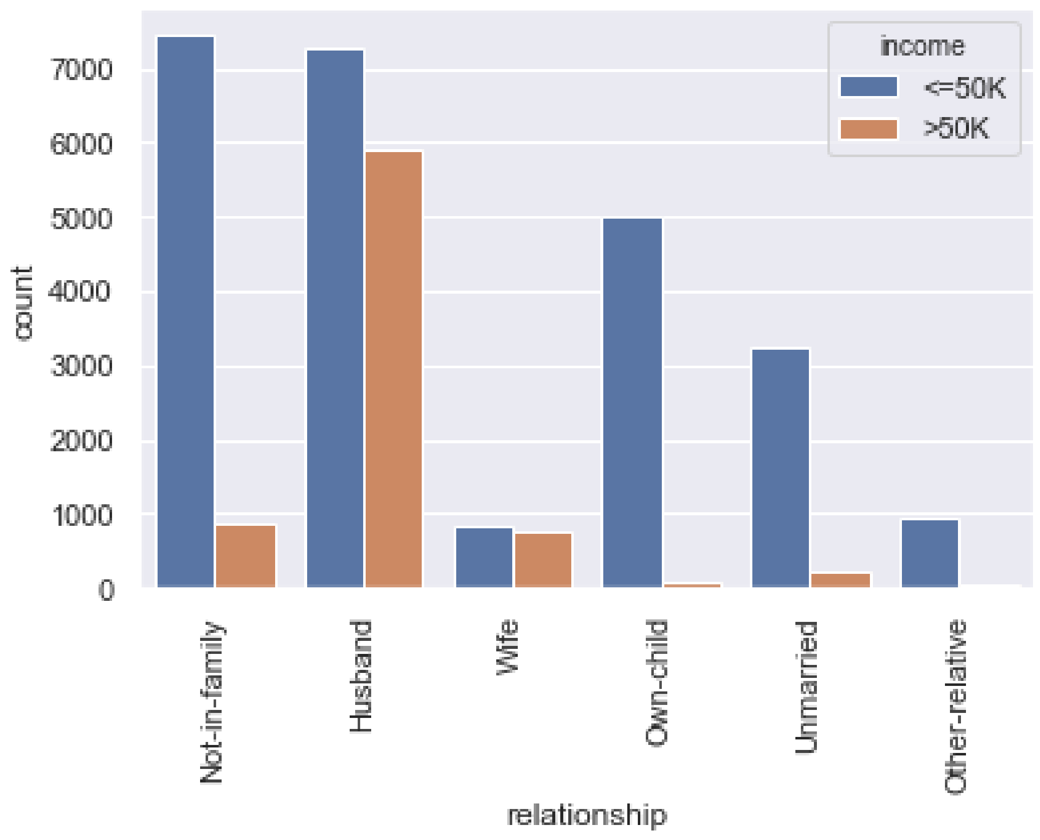
After visualization, slight modification of the data is made by combining various services categories together into a general services category.

```
Out[270]: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]),
           <a list of 12 Text xticklabel objects>)
```



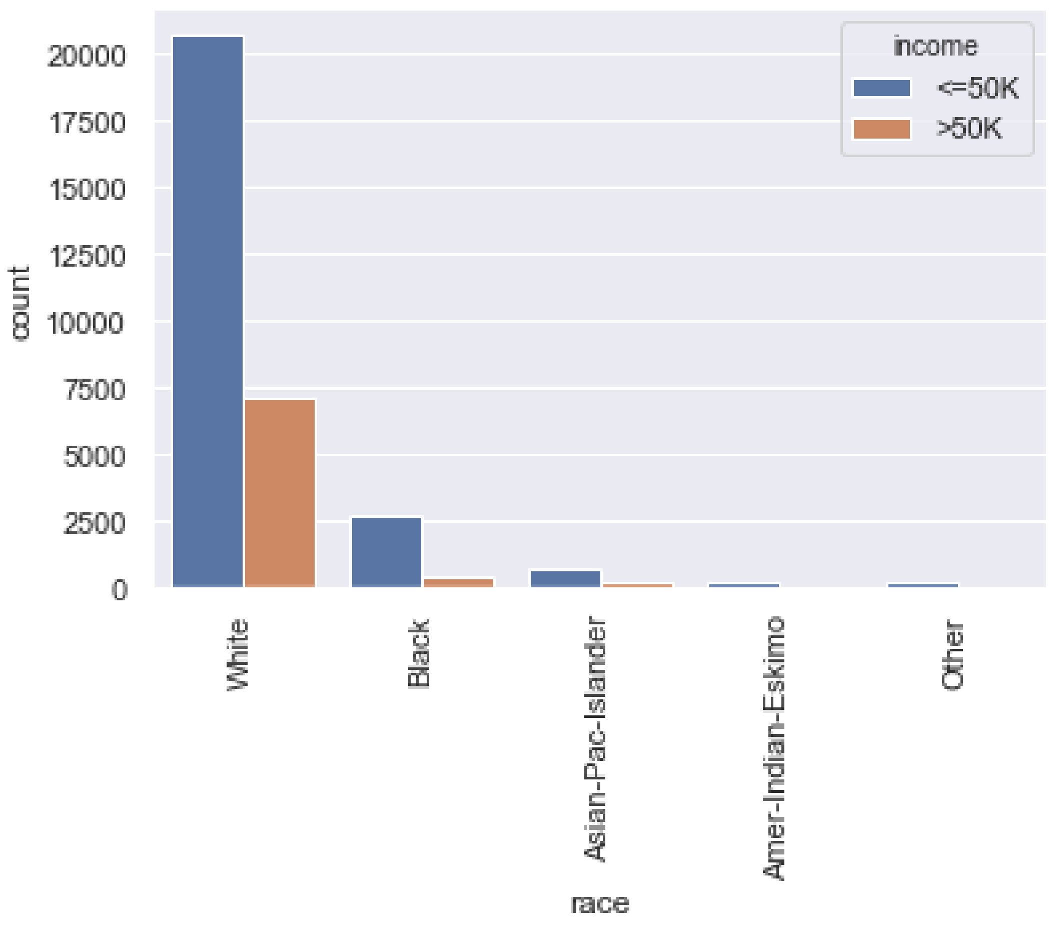
Relationship No modifications were made to this category.

```
Out[271]: (array([0, 1, 2, 3, 4, 5]), <a list of 6 Text xticklabel objects>)
```

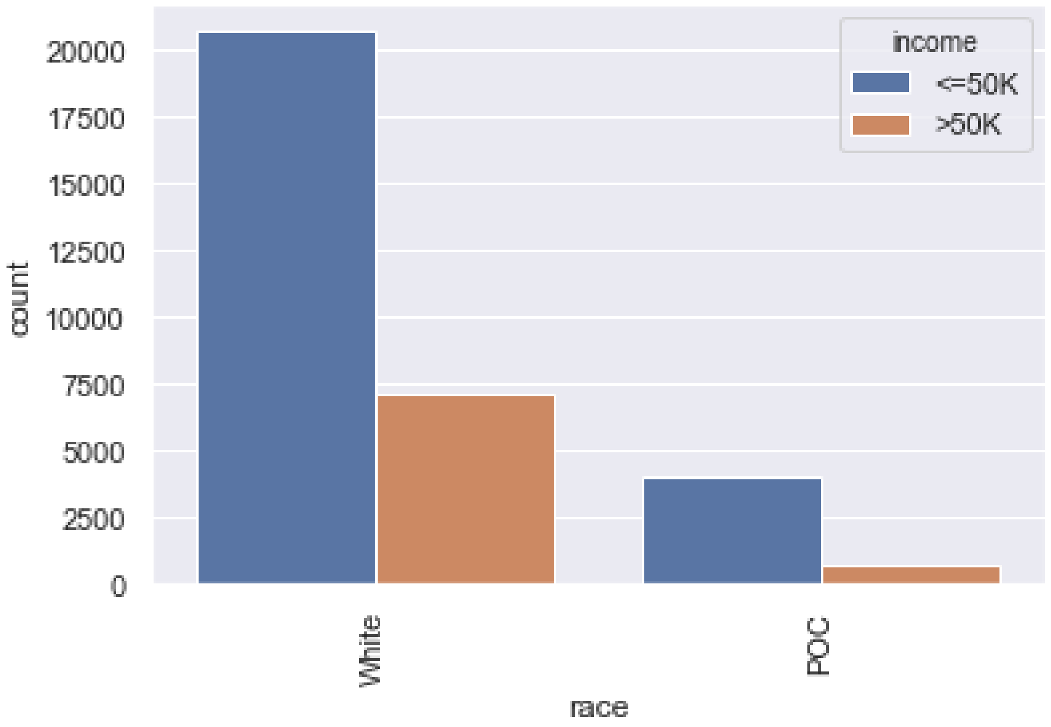


Race The data was modified to combine non-white races into one category, person of color.

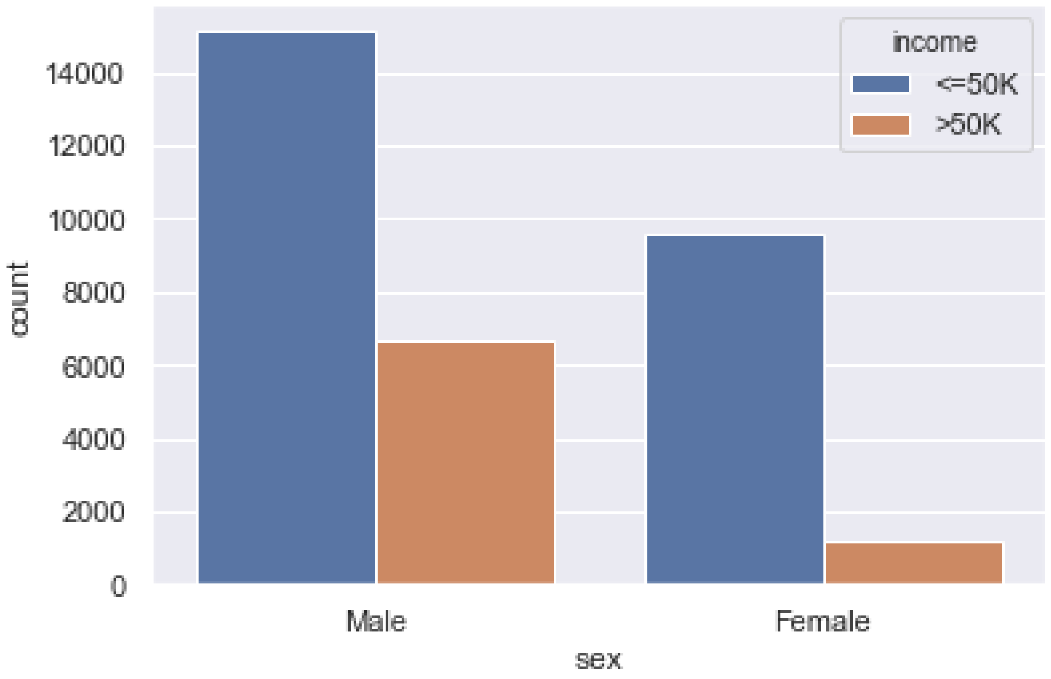
```
Out[272]: (array([0, 1, 2, 3, 4]), <a list of 5 Text xticklabel objects>)
```



```
Out[273]: (array([0, 1]), <a list of 2 Text xticklabel objects>)
```

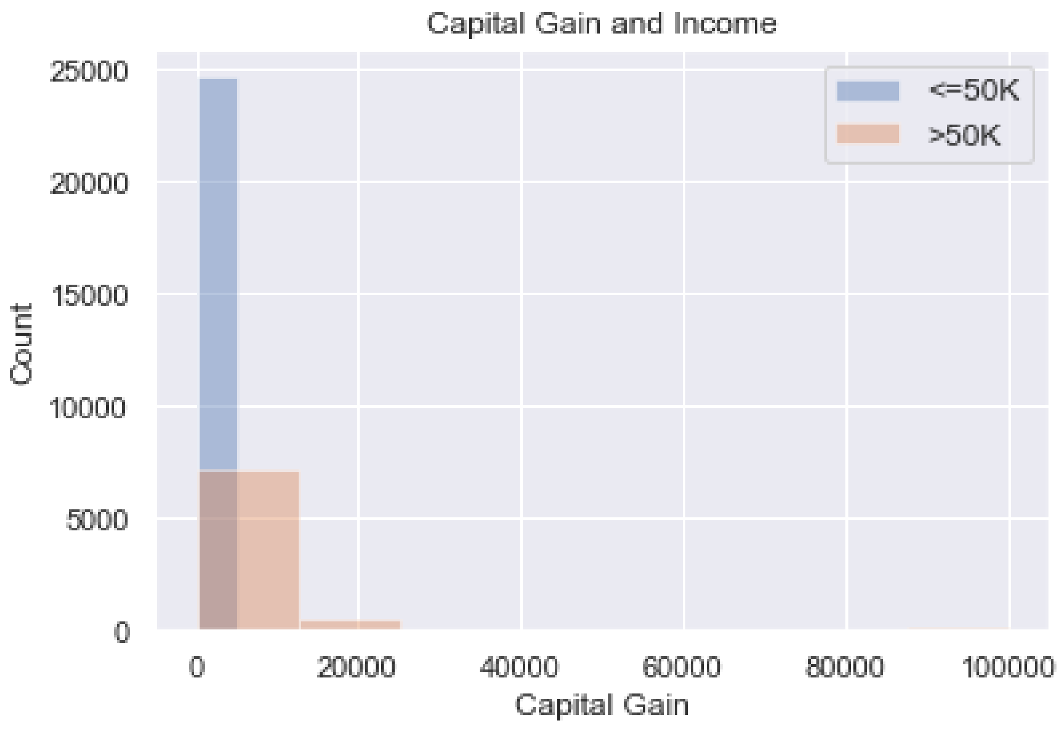


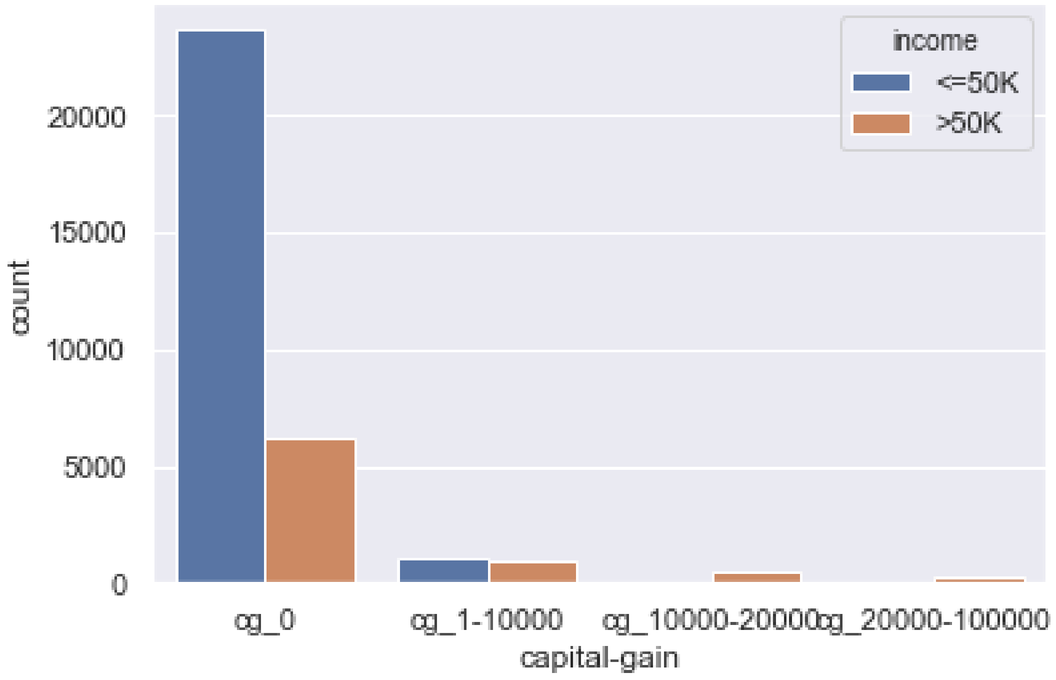
Gender There were no modifications made to this category.



Capital Gains The data was visualized and based on the distribution, binned as shown below

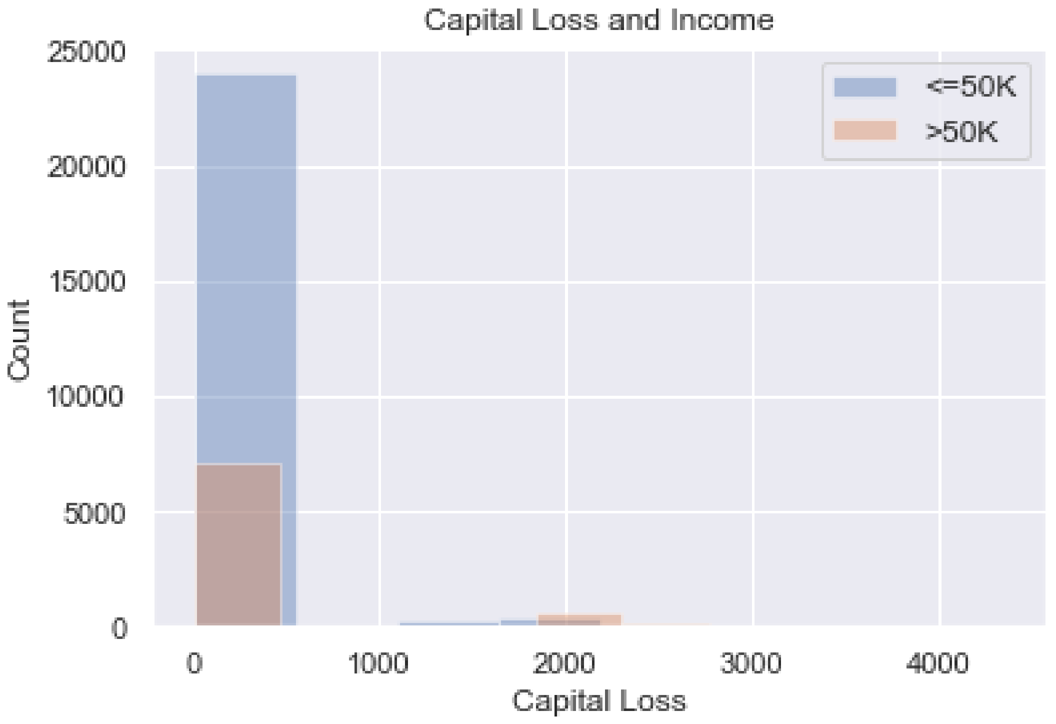
```
Out[275]: Text(0,0.5,'Count')
```

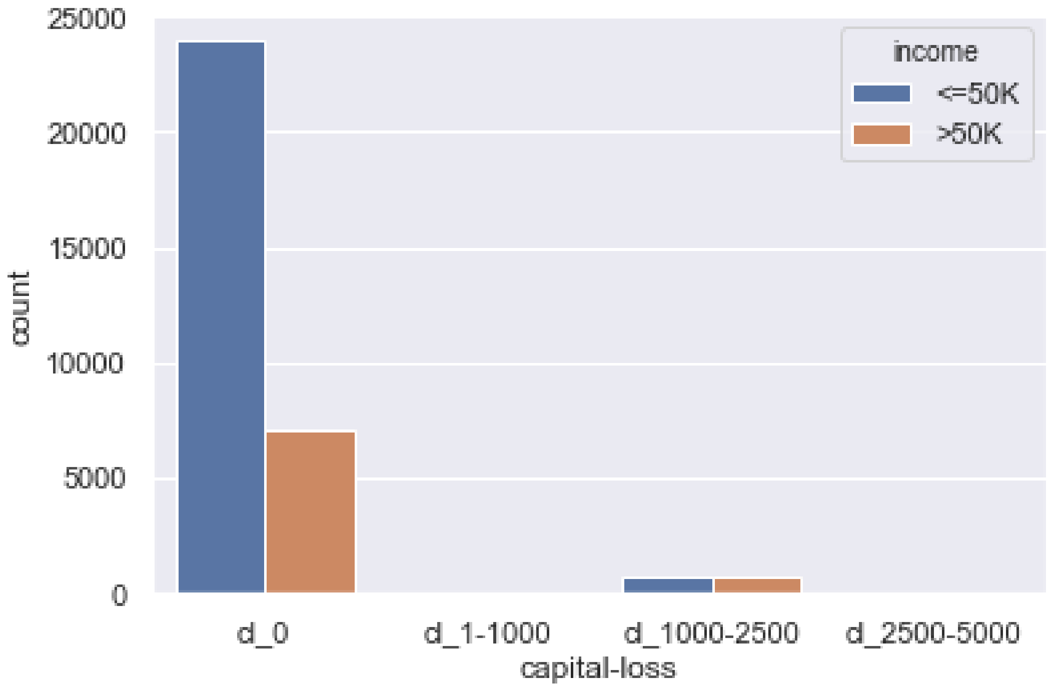




Capital Losses The data was visualized and based on the distribution, binned as shown below

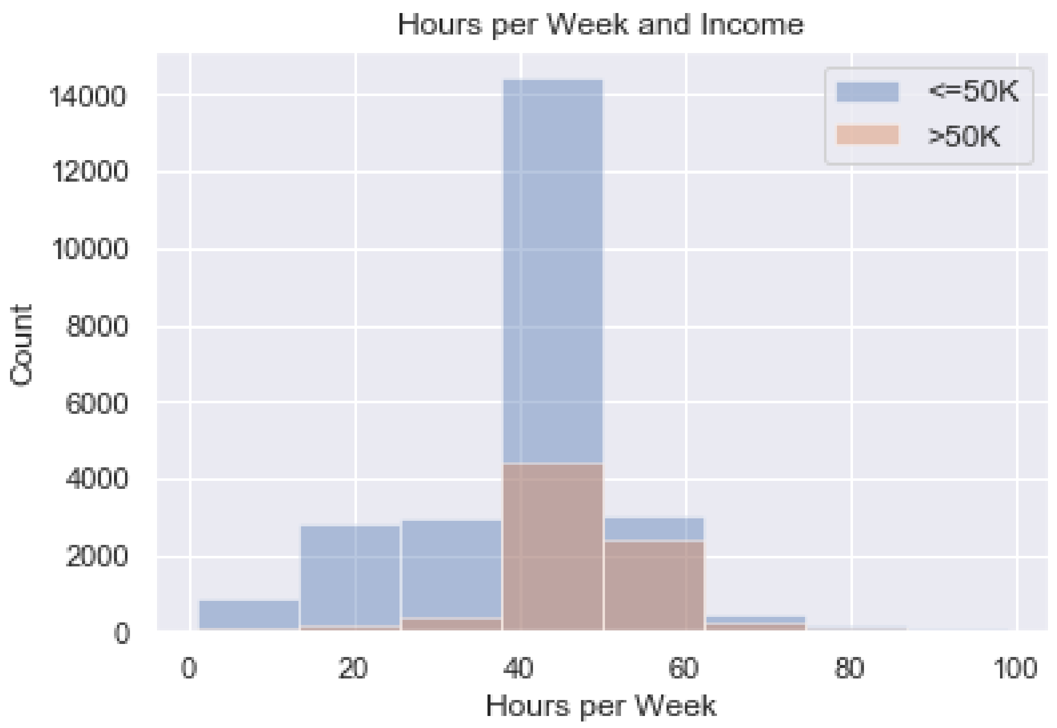
```
Out[279]: Text(0,0.5,'Count')
```

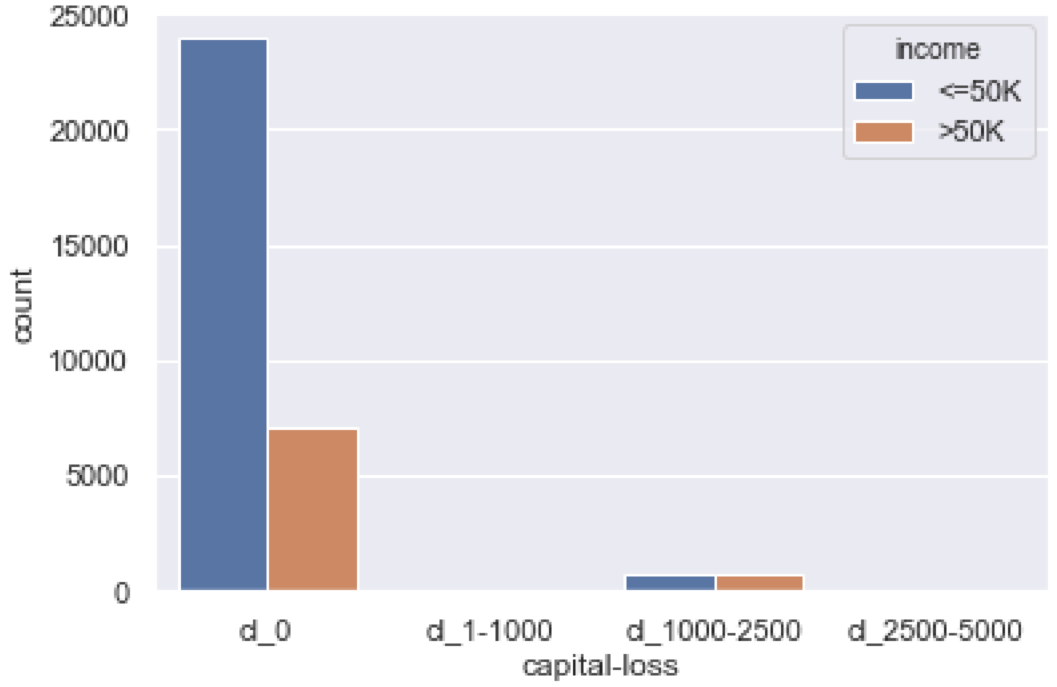




Hours per Week The data was visualized and based on the distribution, binned as shown below

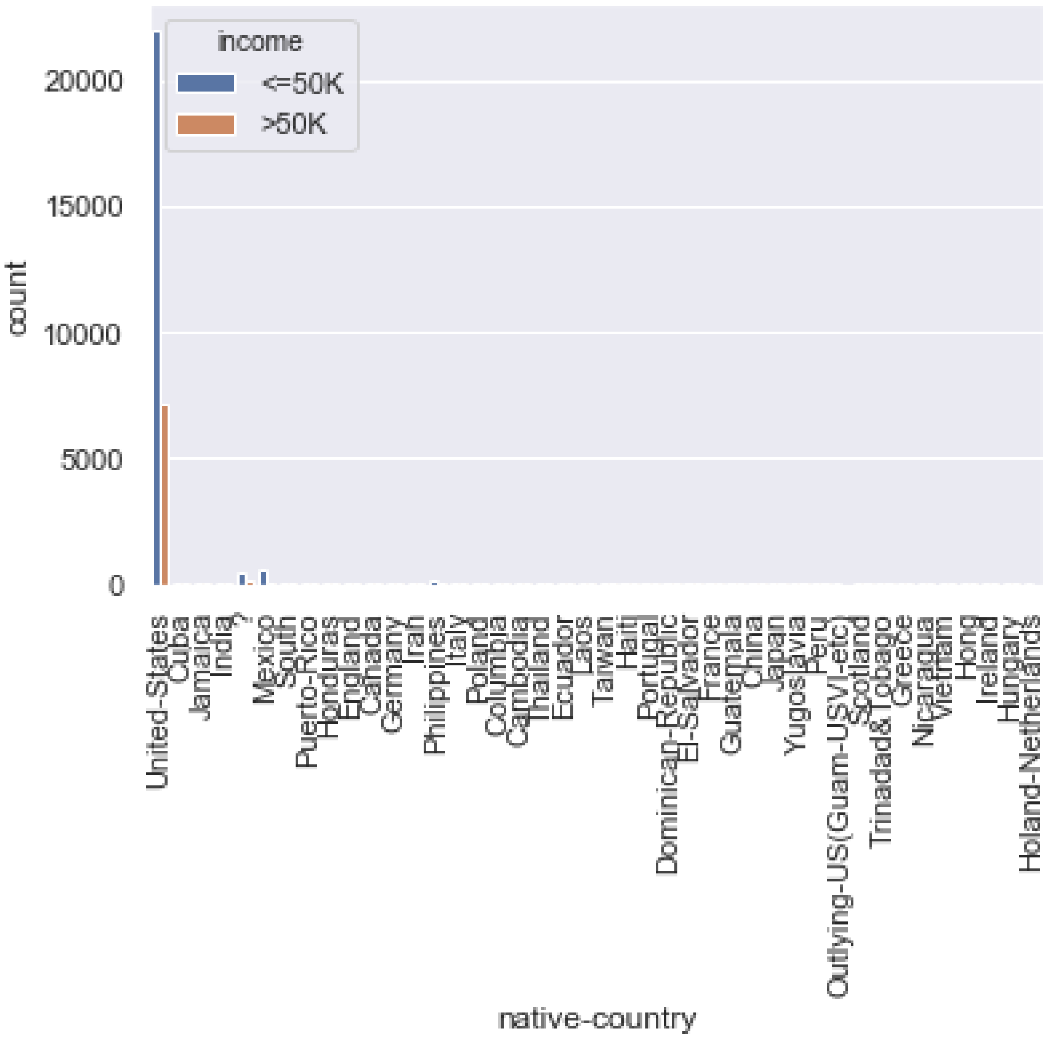
```
Out[282]: Text(0,0.5,'Count')
```

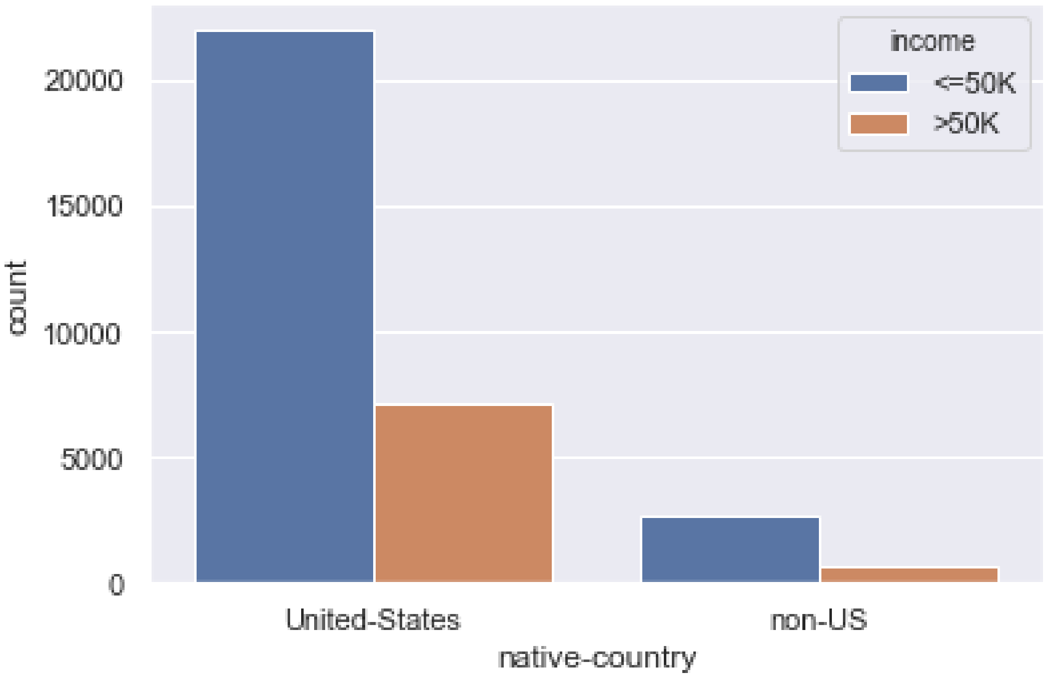

Hours per Week and Income
Count
14000
12000
10000
8000
6000
4000
2000
0
0
20
40
60
80
100
Hours per Week
<=50K
>50K


count
25000
20000
15000
10000
5000
0
d_0
d_1-1000
d_1000-2500
d_2500-5000
capital-loss
income
<=50K
>50K

Native Country The data was simplified into US and non-US country of origin.

```
Out[286]: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                  17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
                  34, 35, 36, 37, 38, 39, 40, 41]),
           <a list of 42 Text xticklabel objects>)
```





After cleaning the data was exported to `data/clean_data.csv`

1.1.2 Step 2: Data Mining

Two different algorithms were used to mine this data set. The Apriori algorithm and the FP-Growth algorithm. An enhancement to the Apriori algorithm that involved partitioning the data. Each implementation is described in detail below.

Apriori Algorithm constructor
The algorithm was implemented as a `class` called `Apriori`. An instance of the class is initialized with three arguments: `data`, a series of transactions in the format: `{t1: [item1, item2, item3], t2: [item2, item4, item3]}`. This is implemented as a Python `dict`, `support_count` the minimum support count, derived from a minimum support threshold and `confidence_pct` the minimum confidence for association mining.

Two static methods can be called to help with preprocessing: `remove_duplicates(data)` which will return the dataset with duplicates removed, and `get_min_support(data, support_pct)` which will calculate the minimum support count based on a minimum support percentage.

instance variables On initialization, all methods to generate the frequent itemsets and association rules are called and stored as class variables. The initialization of `all_frequent` calls the method `find_frequent_itemsets()`. The initialization of `associations` calls the method `generate_all_rules()`. Additional class variables include: - `data`: all transactions stored as a dict - length: the number of transactions in the data set - support_pct: calculated as support_count / length - confidence_pct: passed in as an argument - support_count: passed in as an argument

class methods: finding all frequent itemsets On initialization, a call is made to find_frequent_itemsets. This method generates all frequent itemsets and stores them as a dict in all_frequent. The method generates L1 frequent itemsets by calling generate_F1(). This method first scans the database counting each instance of each item in the dataset and storing them in a dict.

A call in then made to prune_itemsets(candidates) passing in the item count dict as candidates. This method scans the candidates making a list of all items that do not meet the minimum support count. After the list has been generated, the items from the list are removed from candidates and a pruned dict of L1 frequent items is returned. The instance variable storing all frequent itemsets, all_frequent is updated to include this list.

This dict is then passed to the method candidate_generation(frequent). This method first sorts a list (generated from the dict keys) of the items alphabetically. L(k+1) candidates are generated by checking for every combination of the L(k) frequent itemset if (k-1) == (k-1). If this is true, a new (k+1) candidate is generated by combining itemset_1 + the last item of itemset_2. For example, if itemset_1 = ABC and itemset_2 = ABD, the (k+1) candidate is generated as ABCD because (k-1) == (k-1) == (AB), and k (ABC) + k[-1] (D) = ABCD.

While a non-empty candidate set is returned, the candidates that are returned from candidate_generation(frequent) will be pruned with the method candidate_pruning(candidates, freq_data) which generates a list of all (k-1) length subsets. Any itemset of length (k) will have (k) subsets of length (k-1). If any of these itemsets are not in the list of (k-1) frequent itemsets, the candidate is pruned. That is because all subsets of a frequent itemset are also frequent, so if any subset of an itemset is not frequent, that itemset cannot be frequent.

After pruning, the method candidate_count(candidates) is called. This method generates the support count for all candidates of length k post-pruning. The resulting dict is again sent through prune_itemsets(candidates) to eliminate any (k) length itemsets that do not meet the minimum support count.

The all_frequent variable is updated with the candidates that make it through pruning an support counting. These candidates then become the freq_items and the process runs until no additional candidates are generated.

class methods: generating association rules On initialization a call is made to generate_all_rules(). For every frequent itemset in all_frequent a call is made to the method prune_candidate_rules(itemset, count, k). All possible rules are generated by calling generate_candidate_rules(itemset, k) which for an itemset, generates all possible rules: [(s),(l-s)] where s is a subset of the frequent itemset and (l-s) is the complementary set. For example, itemset ABC would generate the following possible association rules: [A->BC, AB->C, AC->B, B->AC, BC->A, C->AB].

A max_count is calculated to determine what the max denominator can be when calculating the confidence, as the itemset count / confidence_pct. Any support count on the right side of the confidence(itemset_1 => itemset_2) that is higher than this max_count is filtered out because the confidence percent would be below the threshold. The association rules that meet the minimum support and confidence thresholds are updated to record these values and added to the associations variable.

FP-Growth Algorithm constructor

The algorithm was implemented as a class called FPTree which utilizes another class called FPNode An instance of the class is initialized with three arguments: data, a series of transactions in the format: {t1: [item1, item2, item3], t2: [item2, item4, item3]}. This is implemented as a Python dict, support_count the minimum support count, derived from a minimum support threshold and confidence_pct the minimum confidence for association mining.

Two static methods can be called to help with preprocessing: remove_duplicates(data) which will return the dataset with duplicates removed, and get_min_support(data, support_pct) which will calculate the minimum support count based on a minimum support percentage.

instance variables On initialization, all methods required to generate the frequent itemsets and association rules are called and stored as class variables. The initialization of f1_sorted calls the method generate_F1_sorted(). The initialization The initialization of associations calls the method generate_all_rules(). Additional class variables include: - data: all transactions stored as a dict - length: the number of transactions in the data set - support_pct: calculated as support_count / length - confidence_pct: passed in as an argument - support_count: passed in as an argument - head: the head of the tree, initialized as an empty FPNode - current_node: keeps track of current node during looping/recursion - pruned_candidates: used to store candidates to be added to prefix path - f1_sorted: sorted list of frequent itemset candidates - node_links: table of lateral links between nodes of the same item - tree: stores the head of the tree - all_frequent: stores all frequent itemsets as they are mined - itemset: current itemset - prefix: tracks the prefix path as the fp-tree is mined recursively
- cp-tree: stores the conditional pattern
- associations: list to store all associations that meet min support and confidence

class methods: finding all frequent itemsets On initialization, a call is made to generate_F1_sorted(). This generates an initial count of items in the transaction data. Items that do not meet the support count are pruned, and a list of items from most frequent to least frequent above the minimum support threshold is generated and returned.

A call to generate_node_links() takes this list and generates a dict that serves as a header table to keep track of the lateral links between nodes of the same item in the FPTree.

Next a call is made to process_all_transactions(). This generates the FPTree. The current node is set to the head of the tree. Each transaction is processed as follows, calling process_transaction(): sort the itemset in the order of most frequent to least frequent, using the F1_sorted list. Process each item in the sorted itemset, calling process_node(). If there is already a child node that contains this item, increment it. If not, create a new FPNode, set the current node as it's parent, add the new link node to the header table, add the new node to the list of child nodes in the current node, then advance the current node to the new node. This builds the full FPTree when all transactions are processed.

A call to mine_all_data() recursively mines the FPTree in a depth-first manner to generate all frequent itemsets. For each item in the F1_sorted list, processing in reverse order, the item is added to a prefix holder variable. The head node is set to the first node associated with that item in the header table, node_links. A new tree is built based on the conditional pattern database generated from that node, the most infrequent of the frequent set, utilizing the method `build_data_from_node()`. Once the new tree is built, that tree is scanned for frequent items and any items that meet the threshold are added to `all_frequent` as the `prefix + item` pattern. For example if the tree is two steps into recursion, with a prefix of `(milk, eggs)` and the new tree build based on the conditional pattern for that prefix shows a count greater than minimum support for `(butter)`, the frequent itemset `(milk, eggs, butter)` is added to the frequent itemset. After `(milk, eggs)`. A conditional pattern is mined for `(milk, eggs, butter)`, any items that meet the threshold in that conditional data base are added as `(milk, eggs, butter, +item)` and so on.

class methods: generating association rules The data for frequent itemset is stored in the same format as Apriori , so the same method is used to generate associations. See the section above for more detail.

AprioriPartition Algorithm The modification to improve efficiency of the mining algorithm is as follows: split the transaction data into four equal transaction datasets, mine each dataset using a support threshold of `min_support / 4`. Mine the datasets, combine the results and prune based on the min_support threshold.

1.1.3 Step 3: Analysis and Results

We are looking for associations in the pattern `(features) => (income: >50k)`.

Running the analysis at a support level of 6,000 (18%) and confidence level of 30%, filtering by the conditional pattern above, there 1,417 frequent itemsets, and 25,507 associations. Of these, only 7 associations fit the pattern shown above. They are:

Rule generated	Support	Confidence
{Male} => {>50K}	20%	31%
{Married-civ-spouse} => {>50K}	21%	45%
{Male, United-States} => {>50K}	19%	31%
{Male, White} => {>50K}	19%	32%
{Married-civ-spouse, United-States} => {>50K}	19%	46%
{Married-civ-spouse, White} => {>50K}	19%	46%
{Married-civ-spouse, cl_0} => {>50K}	18%	43%

The `main.py` script runs at three different levels for the three different algorithms:

Support	Confidence	Frequent Itemsets	Association Rules
15,000 (46%)	70%	98	444
16,000 (49%)	60%	87	499
16,500 (51%)	50%	81	486

Note that the number of frequent itemsets and association rules generated for AprioriPartitioned is slightly different due to rounding. The output for the vanilla Apriori and FPGrowth are identical.