Computer Science Tripos – Part II – Data Science Practical 5

Predicting Income using the Census-Income (KDD) Data Set

Joseph Marchant (jm2129)

Robinson College

28 November 2018

Data Exploration

Initial Look

Running some typical general data analysis functions gave me some initial insights:

```
census.head()
census.info()
census.describe()
```

Here we learn the number of columns (42), entries (37382), and the types of each entry. We can also see that numerous columns (such as 'class of worker' and 'reason for unemployment') have various missing values. Additionally, we have information of min, average and max values for the numerical columns. These min and max values made it clear that some sort of scaling would be necessary.

The first thing I chose to do was explore each feature individually, as we'd need to handle each in some way. To do this, I did the following things:

- Investigated the cardinality and spread of each categorical feature using

  ```
  census[col_name].value_counts()
  ```

- Investigated the range and distribution of each numerical feature using:

  ```
  census[col_name].hist(bins=k)
  ```

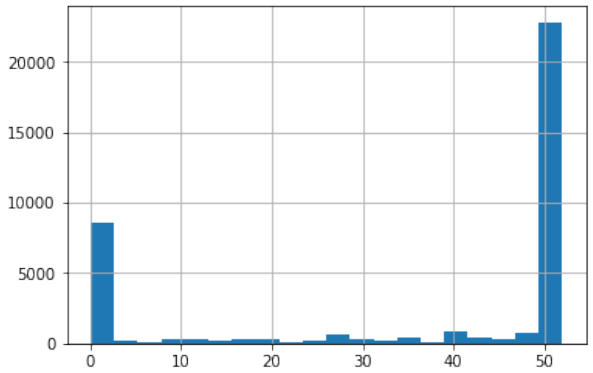
- Use the *census-income.names.txt* file in conjunction with the search feature from *ResearchConnections.org*[1] to extract more information about what each feature represents and how it was collected.

I repeated each of these steps for every relevant column.

[1] https://www.researchconnections.org/childcare/search/variables?q=SEOTR&STUDYQ=04559

Examples and Discussion

For example, many features had tail heavy distributions such as **weeks worked in year**:



For categorical columns, this process highlighted the need for some feature engineering. For example here are the counts for the categories of **education**:

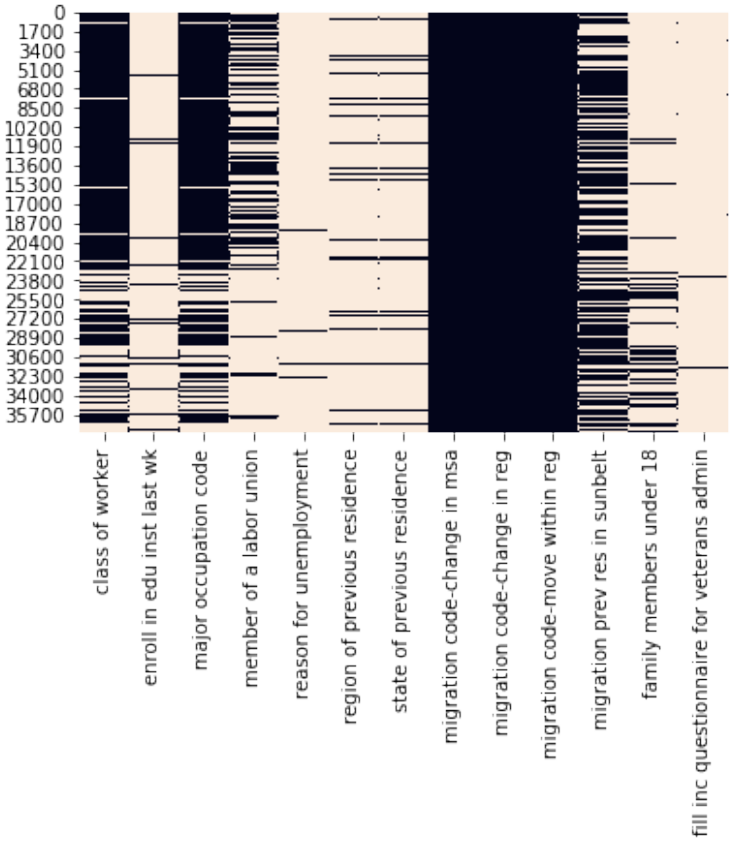
```
High school graduate                    9589
Some college but no degree              6264
Bachelors degree(BA AB BS)              6068
Children                                3740
Masters degree(MA MS MEng MEd MSW MBA)  2566
Associates degree-occup /vocational     1284
10th grade                              1120
Associates degree-academic program      1081
11th grade                              1076
Prof school degree (MD DDS DVM LLB JD)  1056
7th and 8th grade                        998
9th grade                                821
Doctorate degree(PhD EdD)                717
5th or 6th grade                         375
12th grade no diploma                    321
1st 2nd 3rd or 4th grade                 225
Less than 1st grade                       81
Name: education, dtype: int64
```

This demonstrates that there are probably more categories than needed. As well as this, some categories contain too few entries, such as *Less than 1st grade*, and therefore grouping similar categories will allow us to minimise total features whilst maintaining the majority of our prediction accuracy. I show an example of how I reduced the cardinality of this feature in the **Feature Engineering** section.

Some categories even had lots of unknown values, making them potentially worth dropping such as **migration code-change in reg**:

```
?                                  19104
Nonmover                           15478
Same county                         1657
Different county same state          500
Different region                     225
Different state same division        151
Abroad                                73
Different division same region        66
Name: migration code-change in reg, dtype: int64
```

I also plotted a heat-map of the features containing null entries:



This implied that various columns needed to either be dropped or filled in. Since each of these were categorical, there were a few options on what could be causing the missing values and therefore how to handle them:

1. The lack of entries could be representative of an extra category. For example, "Not In Universe", which represents when certain people did not fill this question in.

2. Could have been a data collection error, and we could therefore just fill the remaining values based on the distribution of the values, or similar.

3. Drop the column.

Additionally, some columns were in fact categorical but they had numerical entries. Examples include **year** and **own business or self employed**:

```
0    32150
2     4263
1      969
Name: own business or self employed, dtype: int64
```

Therefore I would need to cast these to discrete categories because as values, any model built would interpret them as comparable values.

Feature Engineering

The above exploration and analysis allowed me to decide upon an action for each column, and a reasoning. Here are my conclusions from the analysis, with an action and reasoning assigned to sets of features.

- – **Action:** No action (note however I do scale these values later).
  - **Reasoning:** No missing values, and a numerical column.
  - **Features:** age, wage per hour, capital gains, capital losses, dividends from stocks, instance weight, num persons worked for employer, weeks worked in year
- – **Action:** Remove column
  - **Reasoning:** Not enough values, or feature not linearly independent from all other data. Or (for race, hispanic origin and sex) we do not want to introduce unethical bias into model.
  - **Features:** industry code, occupation code, race, hispanic origin, sex, region of previous residence, state of previous residence, detailed household summary in household, migration code-change in msa, migration code-change in reg, migration code-move within reg, migration prev res in sunbelt, family members under 18, citizenship, fill inc questionnaire for veteran admin
- – **Action:** Fill in missing values with "Not In Universe", and one hot encode result
  - **Reasoning:** Documentation implies that missing values represent "Not In Universe", and after this change, feature has few categories.
  - **Features:** class of worker, enroll in edu inst last wk, major occupation code, member of a labor union, reason for unemployment
- – **Action:** Reduce cardinality by grouping categories, and one hot encode result
  - **Reasoning:** Many categories overlap when considering their relationship to income, and/or some categories have too few occurrences for a one hot encoding to be sensible.
  - **Features:** education, marital status, full or part time employment stat, tax filer status, detailed household and family stat
  - Here are the new education categories after this change:

```
High School Graduate          9589
College/ Associates Degree    8629
Bachelors Degree              6068
Other                         5017
Post-graduates Degree         4339
Children                      3740
Name: education, dtype: int64
```

- – **Action:** Combine columns, and one hot encode result
  - – **Reasoning:** The features do not hold enough information separately, but combining allows us to encapsulate and separate out the useful information.
  - – **Features:** country of birth father, country of birth mother, country of birth self
- – **Action:** Convert values to categorical column, then one hot encode
  - – **Reasoning:** Numerical placeholders have just been used for categories.
  - – **Features:** own business or self employed, veterans benefits, year, income (note: we do not one hot encode the target variable)
- – **Action:** Just apply one hot encoding
  - – **Reasoning:** Either few enough categories for just one hot encoding to work well, or none of above actions feasible/ necessary.
  - – **Features:** major industry code, live in this house 1 year ago

All the code to complete this feature engineering is inside the **feature_selection** notebook. After this, the data set consisted of 107 columns.

Feature Selection Using Permutation Importance

The main advantage to reducing the number of columns is two-fold: Firstly, it improves interpret-ability of the model and results, as each prediction is based on less information. Secondly, it allows our models (especially Neural Networks) to run faster.

In order to further reduce the number of columns, I chose to apply a technique called **Permutation Importance**. The technique works by building a model, and then for each feature we randomly permute the values in order to evaluate its effect on the prediction. This allows us to rank the features by *Feature Importance*.

I first performed a *StratifiedShuffleSplit* on the data set with a train:test ratio of 4:1. I then applied the *sklearn.preprocessing.StandardScaler* (which performed better than MinMaxScaler by around 1% on each model) to each data split separately (as in the real world we won't be able to scale inputs by some universal mean and variance etc.).

I then fit the data to a *LogisticRegression* model.

On the training set, this model was already performing well:

```
In [8]: from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

        accuracy = accuracy_score(census_target, predictions)
        precision = precision_score(census_target, predictions)
        recall = recall_score(census_target, predictions)
        f1 = f1_score(census_target, predictions)
        print(accuracy,precision, recall, f1)

    0.8899849523491055 0.8400328981186388 0.8249369005552751 0.8324164629176855
```

I then used the *eli5.sklearn* package *PermutationImportance* to rank the features by importance. This returned a table of the following format containing all 107 features:

Weight	Feature
0.0284 ± 0.0058	weeks worked in year
0.0174 ± 0.0033	wage per hour
0.0162 ± 0.0029	education_Post-graduates Degree
0.0157 ± 0.0024	capital gains
0.0129 ± 0.0028	age
0.0078 ± 0.0010	stock dividends
0.0071 ± 0.0014	major occupation code_Executive admin and managerial
0.0068 ± 0.0013	education_Other

The weights represent how much the prediction **worsened** after randomly permuting the values for that column. The variation collected from completing this test multiple times is also shown. This ranking of the features showed that a large chunk weren't helping at all with predictions. Some features even predicted more accurately when randomly permuted (demonstrated by a negative weight in the permutation importance table), such as the categories Yes and No from the **veterans benefits** feature.

Therefore we can see that **weeks worked in year**, **wage per hour** and also **education_Post-graduates Degree** are the best predictors of income. This is a very intuitive result, as we'd certainly expect those working more weeks with better pay per hour and a higher level of education, to be earning more.

From this and multiple iterations, I was able to confidently remove over half the features without a significant decrease in prediction accuracy. More specifically, the data set was reduced to only contain 51 columns (including the target variable), with the following new performance metrics:

In [15]:
```
new_accuracy = accuracy_score(new_census_target, new_predictions)
new_precision = precision_score(new_census_target, new_predictions)
new_recall = recall_score(new_census_target, new_predictions)
new_f1 = f1_score(new_census_target, new_predictions)
print(new_accuracy, new_precision, new_recall, new_f1)
```

```
0.8877779635512456 0.8367095115681233 0.8215042907622413 0.8290371879775853
```

ML Algorithms

Note that the entire finalised data transformation process is stored inside the notebook named **final_full_data_transformation_process.ipynb**. To begin, here is the data we have:

```
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape, X_dev.shape, y_dev.shape)
```

```
(23924, 50) (23924,) (7477, 50) (7477,) (5981, 50) (5981,)
```

I started by creating a generic evaluation function for comparing ML algorithms. This function predicted on the training set, just to give me an initial idea of which algorithms work best. I then built multiple models and evaluated each:

- **Logistic Regression** with *sklearn.linear_model.LogisticRegression*

```
Accuracy: 0.8853033186204099
Precision: 0.8327338129496403
Recall: 0.8180212014134276
F1: 0.8253119429590019
ROC (AUC) Score: 0.9479734272463402
Confusion Matrix:
[[14698  1302]
 [ 1442  6482]]
```

- **Perceptron** with *sklearn.linear_model.SGDClassifier*

```
Accuracy: 0.8441722408026756
Precision: 0.792608089260809
Recall: 0.7171882887430591
F1: 0.7530144428249637
ROC (AUC) Score: 0.9096478735487129
Confusion Matrix:
[[14513  1487]
 [ 2241  5683]]
```

- **Naive Bayes** with *sklearn.naive_bayes.GaussianNB*

```
Accuracy: 0.6756819579720629
Precision: 0.5054237065281704
Recall: 0.9702170620898536
F1: 0.6646207045601902
Confusion Matrix:
[[8477 7523]
 [ 236 7688]]
```

- **Perceptron with Kernel Trick** with *sklearn.kernel_approximation.RBFSampler*

```
Accuracy: 0.5758645415054711
Precision: 0.3834050141613343
Recall: 0.4612569409389197
F1: 0.41874319757117484
ROC (AUC) Score: 0.5581412559944472
Confusion Matrix:
[[10122  5878]
 [ 4269  3655]]
```

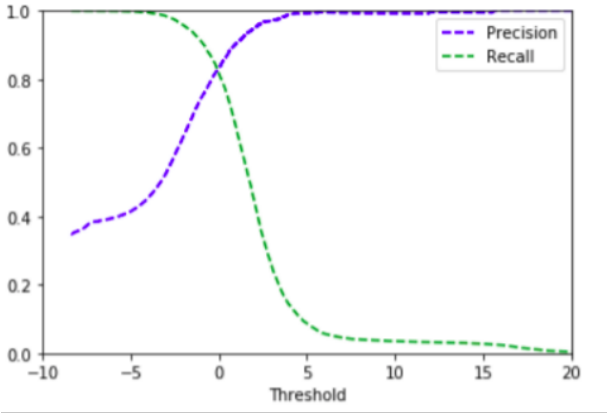
This clearly shows that Logistic Regression is by far the best predictor.

Further Evaluation of Logistic Regression Model

After choosing the best model, I decided to evaluate the performance in more depth and begin to think about what sort of predictor we want.

Precision-Recall Curve

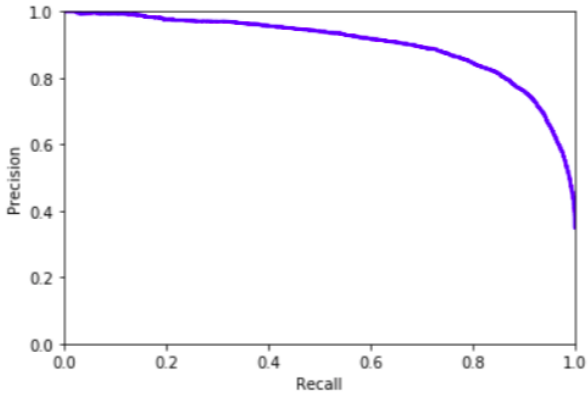
Here is the precision-recall curve for the logistic regression training predictions:



As you can see, the recall has a general tendency of decreasing when you increase the threshold. I.e. the more conservative the classifier becomes (because we increase the threshold required to predict in the positive direction), the more instances of income=1 it is likely to miss. On the other hand, the precision is the opposite: it has a general tendency of increasing with the threshold. I.e. the riskier the classifier becomes, the higher chance a prediction of income=1 is correct.

Precision vs Recall

Here is the plot for precision vs recall from the logistic regression prediction:

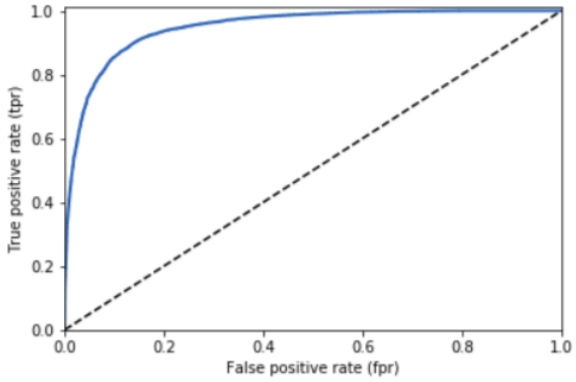


This shows a very predictable relationship between precision and recall: As we try to increase the recall, the precision decreases. Therefore the optimal point, in the case

where we do not prefer either precision or recall, is easy to identify in the top right of the graph. Additionally, if we decided to increase the beta value for the F_beta score (in order to put more emphasis on recall than precision) or visa-verse, we'd see this same phenomenon.

The Receiver Operating Characteristic (ROC) curve

Here is the plot of the ROC curve for the logistic regression predictor:



Here we see a fairly large *area under the curve (AUC)*, demonstrated by the large "ROC (AUC) score" of 0.9479 with the logistic regression model. Therefore the model predicts more positive values correctly than incorrectly.

Neural Networks

I then decided to try out neural nets using *Tensorflow*. All the models built and tested out are inside the notebook **census_data_models.ipynb**. I tried changing various variables for the network, such as:

- Hidden Layer Size (ranging anywhere from 5 upwards)
- Learning Rate (0.05-10)
- Activation Functions
  - *tf.tanh*
  - *tf.sigmoid*
  - *tf.nn.relu*
  - *tf.nn.softmax*
- Number of Epochs (ranging anywhere from 50 upwards)
- Dropout chance (0-1)
- Optimizers
  - *tf.train.GradientDescentOptimizer*
  - *tf.train.AdadeltaOptimizer*
  - *tf.train.AdamOptimizer*

Best Model

- Hidden Layer Size = 100
- Learning Rate = 0.1
- Activation Function = *tf.tanh*
- Number of Epochs = 100
- Dropout chance = 0.5
- Optimizer = *tf.train.AdamOptimizer*

Evaluation of Best Model

The model described in the previous section gave the following output (just added test set accuracy for final run, was previously using the dev set to ensure I wasn't overfitting to the training data):

```
Accuracy after epoch 0 = 47.43772%
Accuracy after epoch 10 = 68.95586%
Accuracy after epoch 20 = 77.57064%
Accuracy after epoch 30 = 82.82896%
Accuracy after epoch 40 = 82.84985999999999%
Accuracy after epoch 50 = 83.16335000000001%
Accuracy after epoch 60 = 82.90002%
Accuracy after epoch 70 = 83.18425%
Accuracy after epoch 80 = 83.15080999999999%
Accuracy after epoch 90 = 83.12573%
Accuracy on dev set: 89.86792
Accuracy on test set: 89.00628999999999
Time taken = 4.79338 seconds
```

Firstly, there's a large disparity between the accuracy on the training data at the last epoch, and the accuracy on the dev and test sets. This is due to the **Dropout**. Furthermore, the conditions are different when testing on the training set vs the other sets - namely the dropout chance is set to 0 for the test and dev sets. Therefore we're able to use all the nodes to make a prediction, which vastly improves the accuracy.

Secondly, the number of epochs being low is a way of implementing **early stopping**. This was because although I could get the training set accuracy as high as 93%, the dev set would perform much worse at around 86%. This is due to the NN overfitting to the training data, and therefore by early stopping we don't allow the NN to overfit.

Here are the key evaluation scores on the test set for the final NN:

```
Accuracy: 0.8900628594356025
Precision: 0.8340734759790068
Recall: 0.8340734759790068
F1: 0.8340734759790068
Confusion Matrix:
[[4589  411]
 [ 411 2066]]
```

The accuracy score means that out of all the NN's predictions, 89.00% of them are correct. The precision score shows that out of all the times the NN predicted "income=1" (positive), 83.41% of them were correct. The recall score represents that the percentage of all cases where "income=1" that the NN correctly classified was 83.41%

The precision score being equal to the recall score implies that the number of **False Positives (FP)** is equal to the number of **False Negatives (FN)**. I.e. whenever we get a prediction wrong, there's a 50% chance it was either a positive or negative prediction.

The preference of precision vs recall would depend on what we use this predictor to do, but as there seems to be no real risk associated with either False Positives (incorrectly predicting someone has a high income) and False Negatives (incorrectly predicting someone

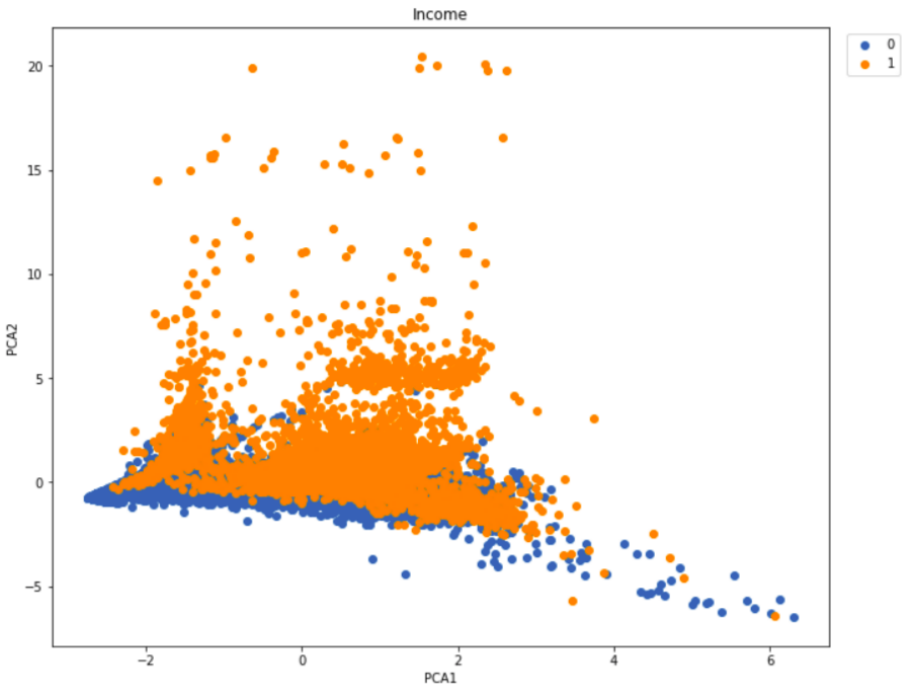
has a low income), a good balance between both seems optimal in favour of maximising accuracy.

Dimensionality Reduction

This section mixes the use of two dimensionality reduction schemes: **Principal Components Analysis (PCA)** and **stochastic neighbourhood embedding with the t distribution (t-SNE)**.

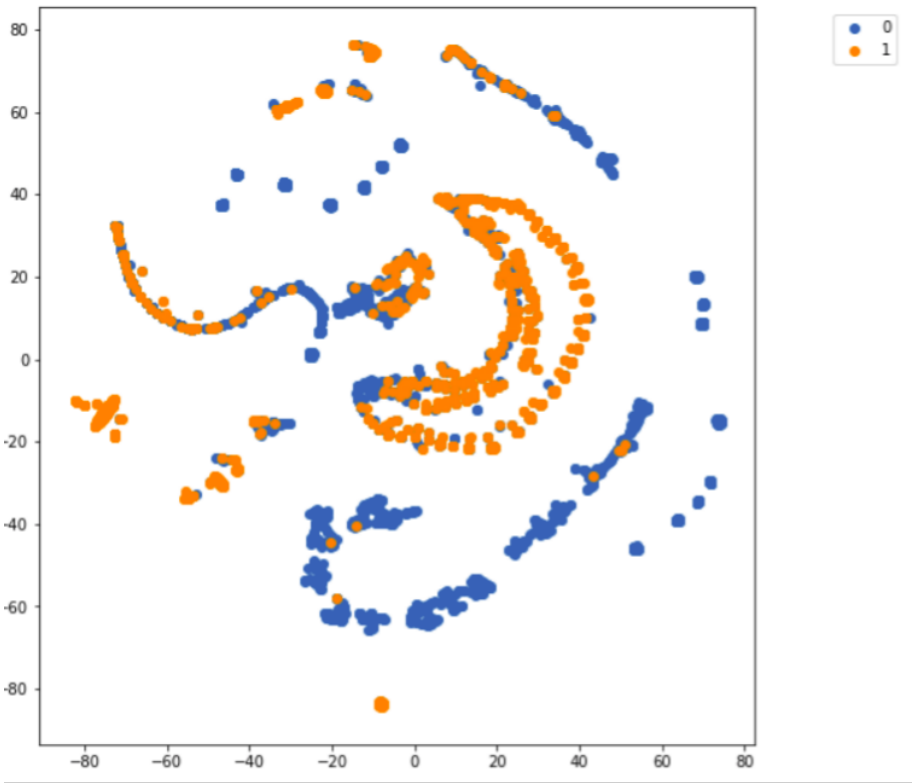
First I began by selecting the numerical features from the dataset.

Using the "median" strategy Imputer (*sklearn.preprocessing.Imputer*) to fill in missing values, and re-scaling the features so they have the same variance, the following plot is produced:



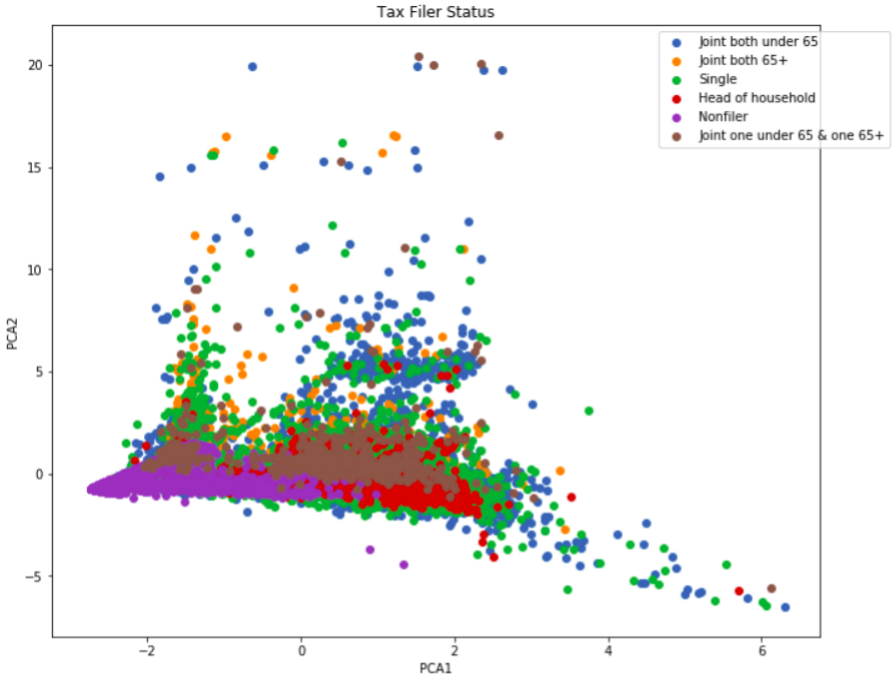
Here we can see that we struggle to separate the income clusters. They overlap a lot, however there is a slight indication of separate clusters formed.

Applying t-SNE to the same scenario gives the following plot:

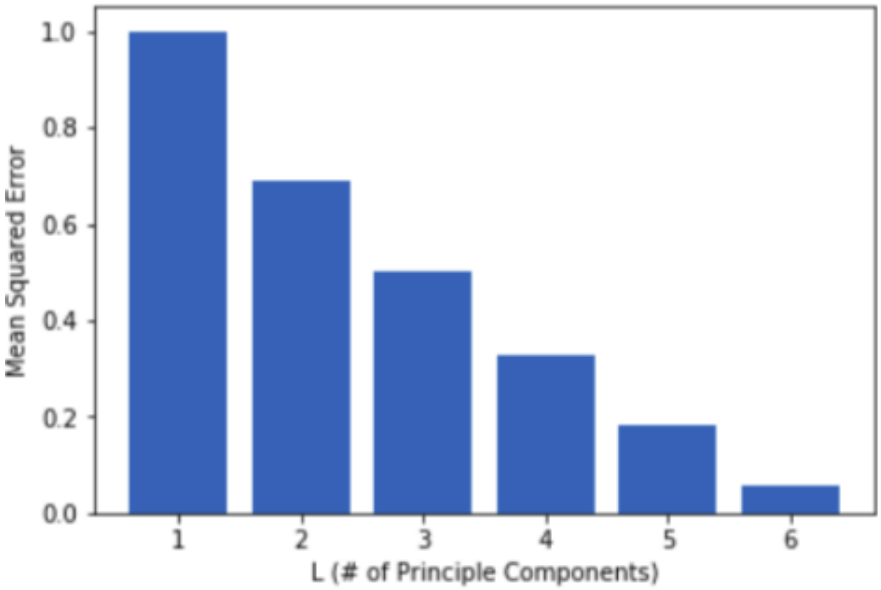


Again there is a lot of overlap, but this time we seem to have a more promising representation of various clusters.

I then tried each of these plots with various categorical columns, with no luck. The plots still had considerable overlap between categories, for example:



We can also plot the number of principle components against the mean squared error (MSE) for PCA as below:



As expected, we see an inverse relationship between MSE and L: as L increases, the MSE decreases. However, the MSE does not decrease as much as we'd hope. I.e. we'd ideally like the MSE to be much lower for when L=2. This is because we'd then be able to predict much more accurately with only 2 principle components, and therefore the graphs above would show clusters more separated with less overlapping. Since this is not the case, 2 principle components simply isn't enough to showcase the clusters in the data well enough.