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

Introduction

This data set contains weighted census data extracted from the 1994 and 1995 current population surveys conducted by the U.S. Census Bureau.[1]

The task is to build a machine learning pipeline to predict the target value based on the other demographic and employment related variables in the data set. The target value represents the level of income: below $50K (value 0) or above $50K (value 1).

The implementation and report should include steps covering data exploration, the implementation of machine learning algorithms, evaluations, visualizations and dimensionality reduction.

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:

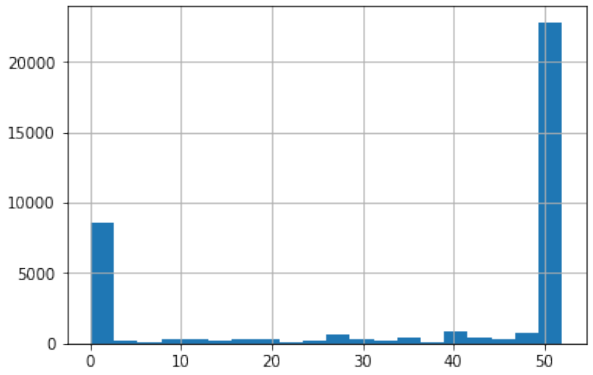
  ```
  census[col_name].hist(bins=k)
  ```

- Use the *census-income.names.txt* file in conjunction with the search feature from *ResearchConnections.org*[2] to extract more information about what each feature represents and how it was collected.

[1] https://archive.ics.uci.edu/ml/datasets/Census-Income+%28KDD%29

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

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



For categorical columns, this highlighted the need for some feature engineering. For example see the following output from the command *census['education'].value_counts()*:

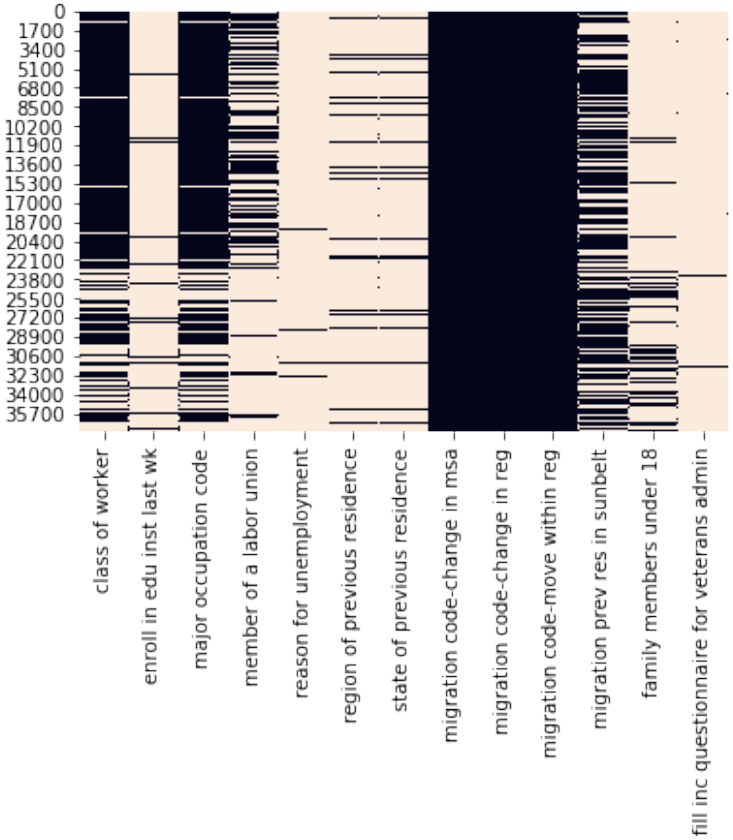
```
High school graduate                 9589
Some college but no degree           6264
Bachelors degree(BA AB BS)           6068
Children                             3740
Masters degree(MA MS MEng MEd MSW MBA)  2566
Associates degree-occup /vocational  1284
10th grade                           1120
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
```

Here we see that there are probably more categories than needed. As well as this, some categories contain far 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 section **Feature Engineering**.

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
```

Another thing worth plotting was a heat-map of the features containing null entries:



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

1. The lack of entry could be an extra category, for example "Not In Universe" which represents when certain people did not need to fill this question in.
2. Could have been a collection error, and we could therefore just fill the remaining values based on either the most frequent category, or based on the distribution of the values given the target variable etc.
3. Drop the column

Additionally, some columns were in fact categorical but they had numerical entries to represent each category. 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 object types. This is because as values, any model built would interpret these as comparable numerical values, but in reality for example for the above example: 1 represents No, and 2 represents Yes, and it makes no sense for "Yes" to be twice the weight of "No".

Feature Engineering

This exploration and analysis allowed me to decide upon an action for each column, and a reasoning (a table for all this is stored in *feature_engineering_sheet* in the code folder uploaded with my project). I used this to group the features by action and reasoning, as follows:

- – **Action:** No action
  - **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) 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 little enough columns for one hot encoding to be feasible.
  - **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 values for a one hot encoding to be sensible. After changes, one hot encoding will be suitable.
  - **Features:** education, marital status, full or part time employment stat, tax filer status, detailed household and family stat

```
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:** Don't 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 numeric values to categorical column, then one hot encode
  - – **Reasoning:** Information not representative of a numerical scale, numerical placeholders have just been used for categories. So better to represent as categories.
  - – **Features:** own business or self employed, veterans benefits, year, income (note: we do not one hot encode the target variable)
- – **Action:** One hot encode, and no other actions
  - – **Reasoning:** Either few enough values for just one hot encoding to work well, or none of above actions sensible/ feasible.
  - – **Features:** major industry code, live in this house 1 year ago

All the code to complete this feature engineering is inside the **census_data_feature_selection.ipynb** notebook, with a showcase of the after-changes data set.

At this point, the dataset consisted of 107 columns.

Feature Selection Using Permutation Importance

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 started by applying the *sklearn.preprocessing.StandardScaler* on the data, to sort the range issues discussed earlier. I then performed a *StratifiedShuffleSplit* on the data set with a train:test ratio of 4:1.

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:

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 text multiple times is also shown. This ranking of the features showed that a large chunk of the features 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.

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 new metrics becoming:

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

Using *StratifiedShuffleSplit* I created a Train:Dev:Test split of the data, with the following shapes:

```
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 of these, in every metric.

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

After trying various combinations of above, the best results came from the following variables:

- 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 on the previous page 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 data):

```
Accuracy after epoch 0 = 56.76308%
Accuracy after epoch 10 = 82.22705%
Accuracy after epoch 20 = 76.40445%
Accuracy after epoch 30 = 82.47367%
Accuracy after epoch 40 = 82.03478%
Accuracy after epoch 50 = 82.66594%
Accuracy after epoch 60 = 82.97943%
Accuracy after epoch 70 = 82.79552%
Accuracy after epoch 80 = 83.12573%
Accuracy after epoch 90 = 83.07139%
Accuracy on dev set: 89.85119999999999
Accuracy on test set: 89.01966
Time taken = 4.90916 seconds
```

Before showcasing some more evaluation statistics, I want to outline some interesting points regarding this NN and its results.

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 predicition, which vastly improves are accuracy.

Secondly, the number of epochs being low was me 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.