Predicting Credit Card Approvals

Paul Coster

✔ **Predict Approval**

✖ **Predict Denial**

?

Abstract

This project was an analysis of the ‘Credit Card Approval’ dataset from the UCI Machine Learning Repository. The aim of the project was to determine which of the features in the dataset are the most important for determining whether a credit card application should be approved or denied, and whether a ML model could accurately predict the result of an application.

The prepared dataset was fitted using **k-Nearest Neighbors**, **Decision Tree** and **Logistic Regression Classifiers**, using a **GridSearchCV** to find some optimum hyperparameters for each classifier type, and finally training and testing each optimised model on 66% and 33% of the dataset, respectively.

It was found when fitting the models on only the numeric features of the dataset, that the application result could be predicted with around **76-78% accuracy** with all the classifiers used, with the most important numerical features being ‘**Income**’ ‘**Credit Score**’, ‘**Years Employed**’ and ‘**Debt**’. However when the models were fit on the numerical and categorical features of the dataset, **the most important feature was found to be ‘Prior Default’**, and a simple Decision Tree Classifier using only the ‘**Prior Default**’ feature achieved **86% accuracy**, **significantly higher than the baseline accuracy of 55% for the dataset.**

Motivation

In the United States alone, many millions of credit card applications are received by commercial banks each year. From 2018-2019, the number of open credit card accounts in the U.S. increased by around 9 million.[1]

A credit card application often requires many details from the applicant, such as Income, Debt, Employment Status, Length of Employment, Marriage Status etc. Each bank will then use a combination of these details to determine whether or not to approve or deny the application. Doing such a process manually is tedious and prone to error, and can instead be automated using machine learning techniques.

Using such machine learning techniques on a labeled dataset of credit card applications, we can determine if certain classification models are more or less accurate at predicting whether an application should be approved or denied. Furthermore we can analyse which application details are the most important for determining if an application is approved or denied, giving us insight into this particular bank's criteria for a successful credit card application.

[1] Credit Card Ownership Statistics (https://www.creditcards.com/credit-card-news/ownership-statistics/)

‘Credit Approval’ Dataset:

The ‘Credit Approval’ Dataset can be found in the UCI Machine Learning Repository.[1] The first usage of the dataset was in **1987** and was described by its contributor as:

‘The data ... were provided by a large bank. Each case concerns an application for credit card’ [2]

The dataset consists of:

- **690 samples** of credit card applications, **each labelled as being ‘approved’ or ‘denied’.**
  - The dataset contains 55% ‘denied’ samples and 45% ‘approved’ applications.
- Each sample consists of **15 features** (**6 numeric** and **9 categorical)**, and its label.
- **The features of the dataset have been anonymized** to protect the privacy of the applicants. However analysis of the values of each feature have lead to some tentative identifications for each of the features, which have been used in this analysis to aid identification of the features[3]:
  - Numeric Features: ‘Age’, ‘Debt’, ‘Years Employed’, ‘Credit Score’, ‘Zip Code’, ‘Income’
  - Categorical Features: ‘Sex’, ‘Marriage Status’, ‘Bank Customer’, ‘Education Level’, ‘Ethnicity’, ‘Employment Status’, ‘Drivers Licence’, ‘Citizenship’

[1] ‘Credit Approval’ Dataset at UCI ML Repo (https://archive.ics.uci.edu/ml/datasets/Credit+Approval)
[2] J.R. Quinlan (Quinlan. "Simplifying decision trees", Int J Man-Machine Studies 27, Dec 1987, pp. 221-234.)
[3] R. Kuhn ‘Analysis of Credit Approval Data’ (http://rstudio-pubs-static.s3.amazonaws.com/73039_9946de135c0a49daa7a0a9eda4a67a72.html)

Data Preparation and Cleaning

Data Cleaning:
The ‘**Credit Approval**’ dataset contains 37 samples with one or more missing values (denoted by ‘?’). These samples were dropped from the dataset, as they only comprised 5% of the data, and the balance of approved and denied applications in the remaining dataset was unaffected.

Data Preparation:
As the dataset had a mixture of numerical and categorical variables, the categorical variables were converted to numerical representations using the Scikit-Learn Label Encoder. Then all the features were standardised using Scikit-Learn Standard Scaler.

It was found during some initial analysis that the numerical values in the ‘**Zip Code**’ feature were somewhat meaningless and did not provide any predictive power during ML tasks (in fact, using this feature made **k-NN** models less accurate), and so this feature was not used in any of the following analysis.

Research Question(s)

1. Which application details (dataset features) are the most important determiners of credit card application approval or denial in the dataset?

2. Can a machine learning classification model predict which applications will be approved or denied, with better accuracy than the 55% baseline?

Methods

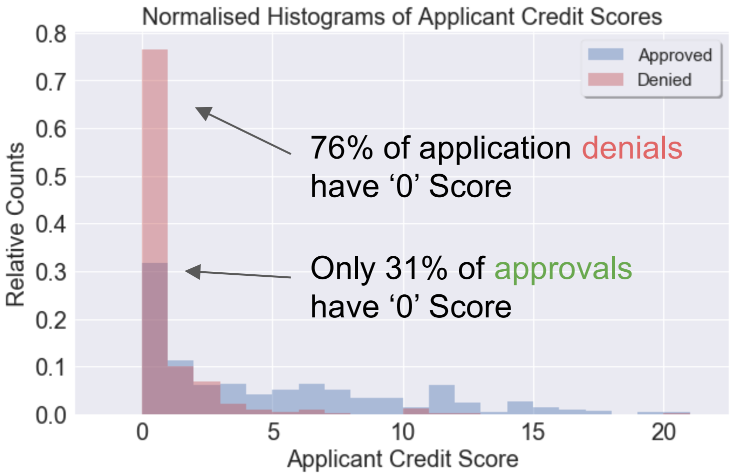
The numerical and categorical variables were first visualised using a mixture of histograms, bar and scatter plots, to determine if any trends in application approval/denial were indicated by specific features. This gave an indication of which of the features were likely to be important for classifying an application as ‘approved’ or ‘denied’

Thereafter, machine learning models were trained and tested, firstly using only the numerical features of the dataset, and then on the numerical and categorical features:

- **Five-fold Grid Search Cross Validation** was used to train and test the classification models used, and find some optimum hyperparameters for the models.
- **k-Nearest Neighbors**, **Decision Tree** and **Logistic Regression Classifiers** were trained and tested on the dataset.
- The position of features in the fitted **Decision Tree Classifier**, and the Feature Coefficients of the **Logistic Regression Classifier** were used to determine the more predictive features of the dataset.
- Each model type with optimised hyperparameters was then trained on 66% of the dataset and tested on the remaining 33%, so that each model’s accuracy, sensitivity and specificity could be compared.

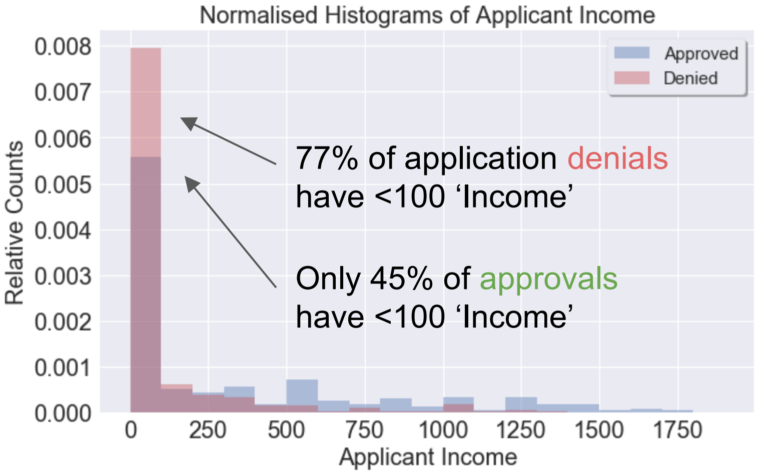
Visualisation of Numerical Features:

- A low ‘**Credit Score**’ or ‘**Years Employed**’ value is an indicator of application denial.
- Applicants with a higher ‘**Income**’ value are more likely to be approved.
- The distributions of ‘**Age**’ and ‘**Debt**’ (not shown) for approved and denied customers were quite similar, suggesting that these features were less important for determining application status.



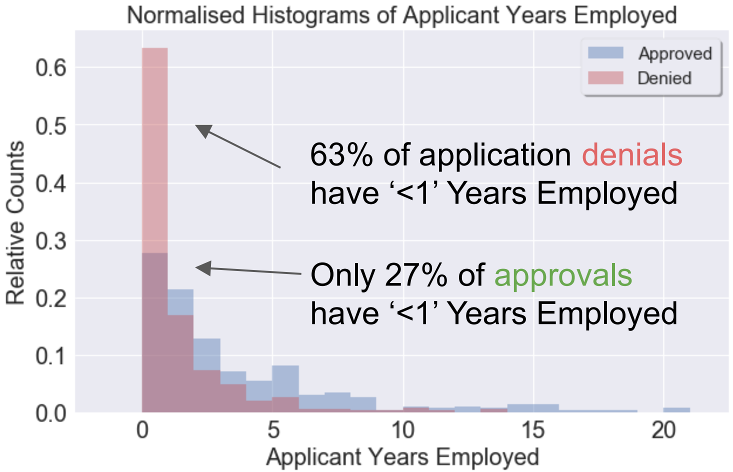
76% of application denials have ‘0’ Score

Only 31% of approvals have ‘0’ Score



77% of application denials have <100 ‘Income’

Only 45% of approvals have <100 ‘Income’

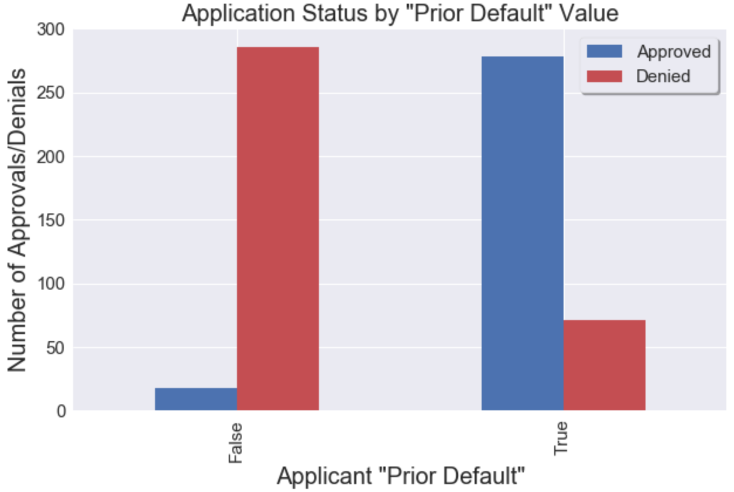


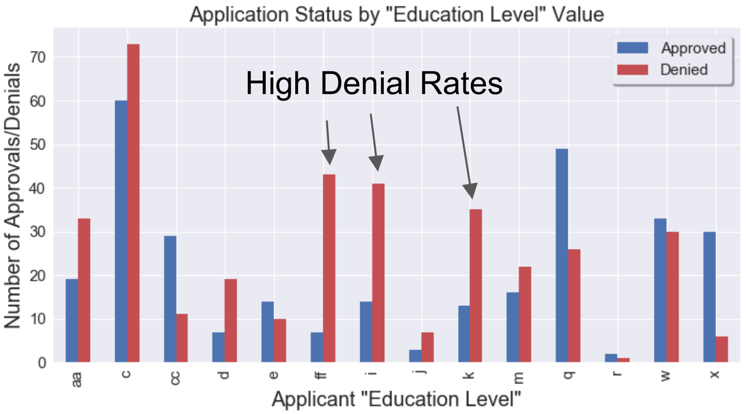
63% of application denials have ‘<1’ Years Employed

Only 27% of approvals have ‘<1’ Years Employed

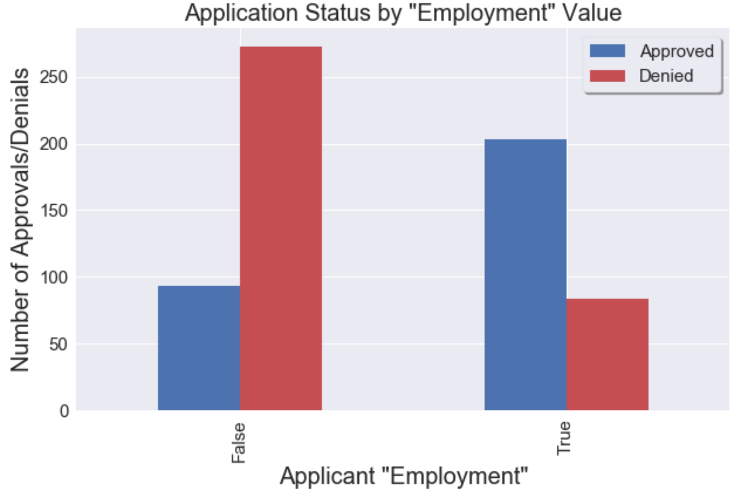
Visualisation of Categorical Features:

- Applicant's '**Prior Default**' value is a very strong indicator of approval or denial, as is applicant's '**Employment**' value:
    - Only 6% of applicants with '**Prior Default**' are approved.
    - 80% of applicants without a '**Prior Default**' are approved.
    - 70% of '**Employed**' applicants are approved.
    - 75% of non-'**Employed**' applicants are denied.
- Some sub-categories of other categorical features also display large differences in approval and denial rates e.g. '**Education Level**' (below), '**Bank Customer**' and '**Marital Status**'.


Application Status by "Prior Default" Value


Application Status by "Education Level" Value

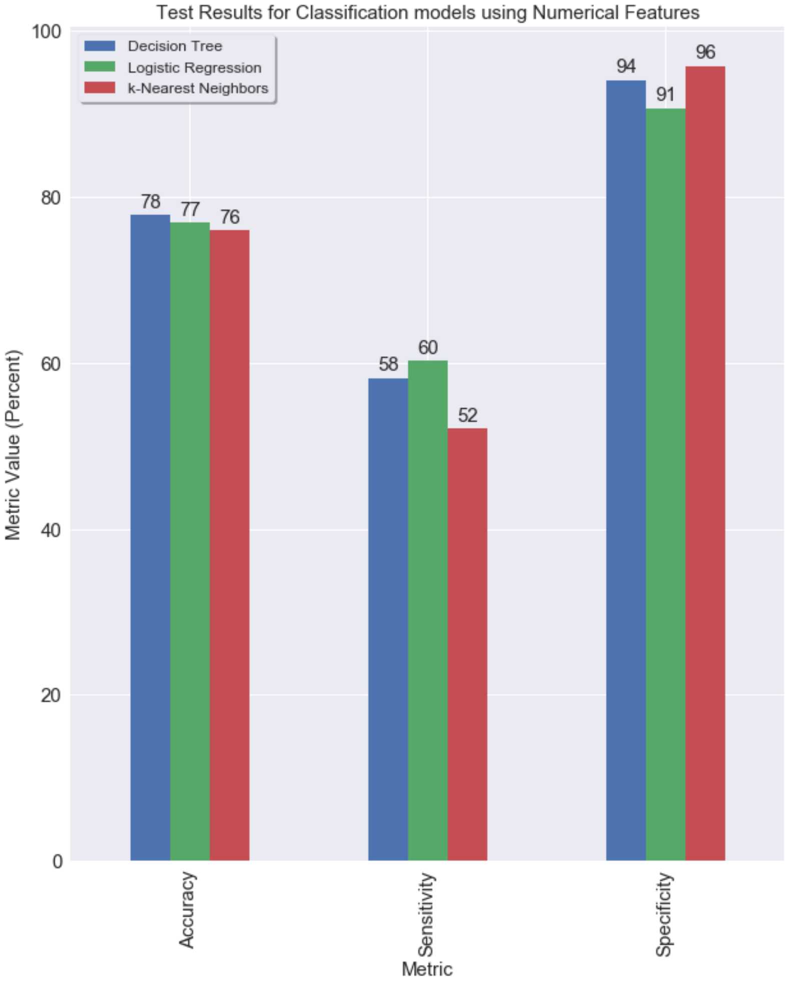
High Denial Rates


Application Status by "Employment" Value

Predicting approvals with ML-Models:

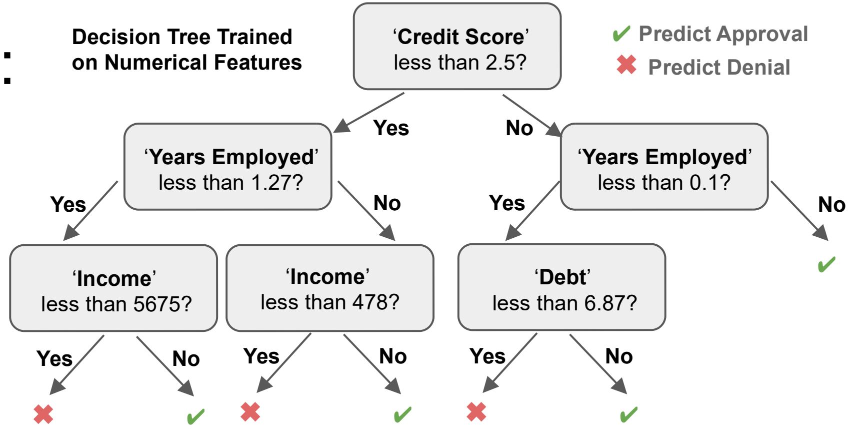
Firstly using only the numerical features:

- **k-Nearest Neighbors**, **Decision Tree** and **Logistic Regression Classifiers** were trained firstly on **only the numerical application details**:
  - Features used were '**Age**', '**Debt**', '**Years Employed**', '**Credit Score**' and '**Income**'.
- The models were found to have **similar overall accuracy** during testing, with the **Decision Tree** having the highest accuracy (78%).
- However, the three models were found to be much better at correctly predicting applications that should be denied (**Specificity** 91-96%) than those that should be approved (**Sensitivity** 52-60%)
- The **k-NN model was found to have the highest specificity (96%)**, which may be preferable despite this model's slightly lower accuracy, if the bank wishes to avoid giving 'bad' customers credit card approval wherever possible.



Most important numerical features:

- The **Decision Tree** structure and the coefficients of the **Logistic Regression** models trained on only the **numerical features**, can give insight as to which of these application details are significant for predicting application approval.
- The Decision Tree model uses **‘Credit Score’**, **‘Years Employed’**, ‘**Income**’ and ‘**Debt**’ features to predict applications as approved or denied.
- The feature coefficients for the **Logistic Regression** Model show that ‘**Income’, ‘Credit Score’** and ‘**Years Employed’** are the most significant application details, with ‘**Debt**’ somewhat less important.
- **Neither the Decision Tree nor the Logistic Regressor** use the ‘**Age**’ parameter to predict approval status, **showing that this feature does not have significant predictive capacity**.



Decision Tree Trained on Numerical Features

✓ Predict Approval
✗ Predict Denial

‘Credit Score’ less than 2.5?

Yes: **‘Years Employed’** less than 1.27?

No: **‘Years Employed’** less than 0.1?

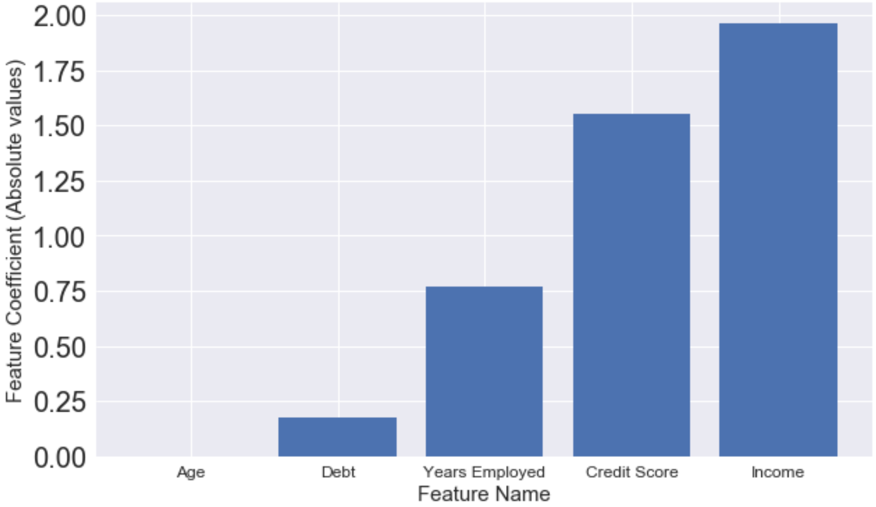
‘Income’ less than 5675? Yes ✗ No ✓

‘Income’ less than 478? Yes ✗ No ✓

‘Debt’ less than 6.87? Yes ✗ No ✓

No ✓

Feature Coefficients for Logistic Regressor



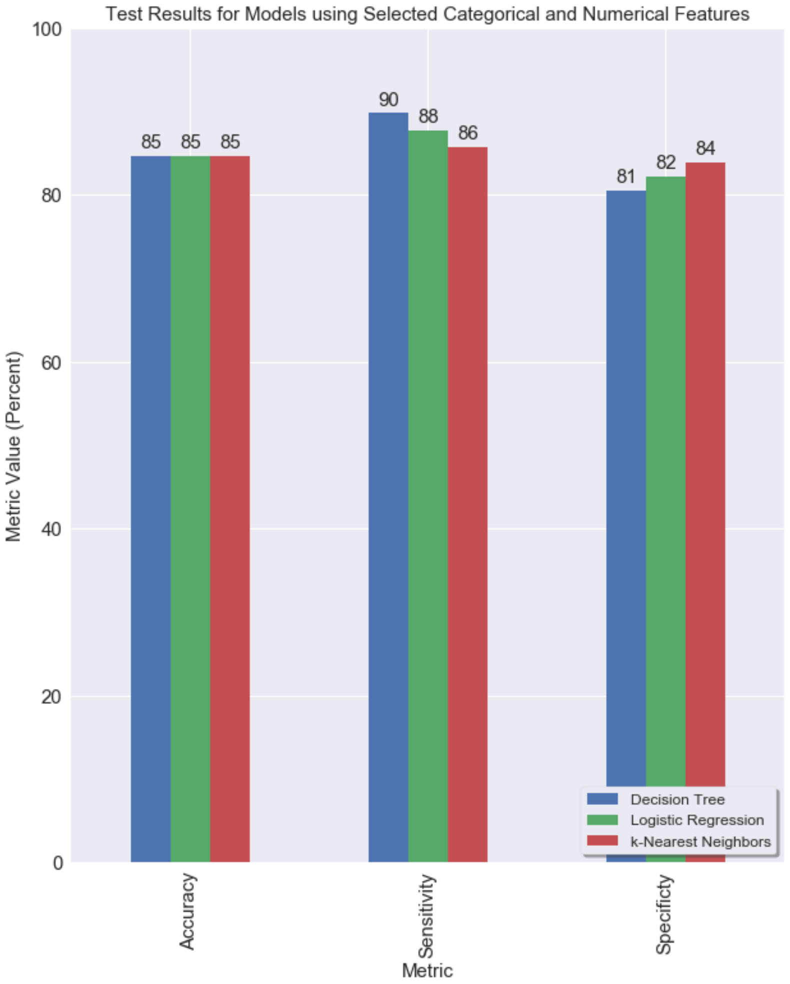
Predicting approvals with ML-Models:

Using selected numerical and categorical features:

- **k-Nearest Neighbors**, **Decision Tree** and **Logistic Regression Classifiers** were then trained on the most predictive numerical and categorical features:
  - Numerical features used were ‘**Years Employed**’, ‘**Credit Score**’ and ‘**Income**’.
  - Categorical features used were ‘**Marital Status’, ‘Bank Customer’, ‘Employment’,** and **‘Prior Default’**
  - Adding more application features to the models was not found to significantly improve accuracy.

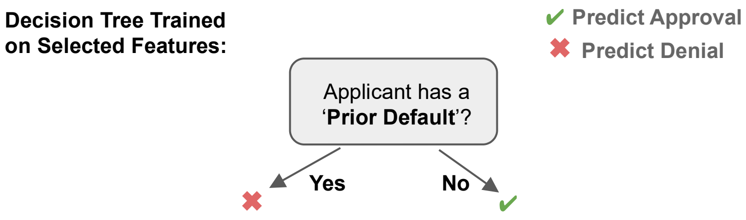
- **All models had an improved accuracy of 85%** using these features. **This accuracy is significantly above the baseline accuracy of 55%.**

- The improvements in accuracy were caused by the models having greatly improved ability to predict applications that should be approved (**Sensitivity** 86-90%), while having a slightly decreased ability to successfully predict applications that should be denied (**Specificity** 81-84%).



Most important selected features:

- Looking at the **Decision Tree** structure, and feature coefficients for the **Logistic Regression**, where both models were trained on the **selected features**:
- A very simple decision tree is found with **85% prediction accuracy**, only using the ‘**Prior Default**’ categorical feature!
- The feature coefficients for the **Logistic Regression** Model trained on the selected features also show that ‘**Prior Default**’ is a highly important feature for this model, as well as ‘**Income**’. The other selected features all have some level of importance to the model.
- Overall these results demonstrate **that applicants with a ‘Prior Default’ would most likely be denied credit approval by this bank.**



Decision Tree Trained on Selected Features:

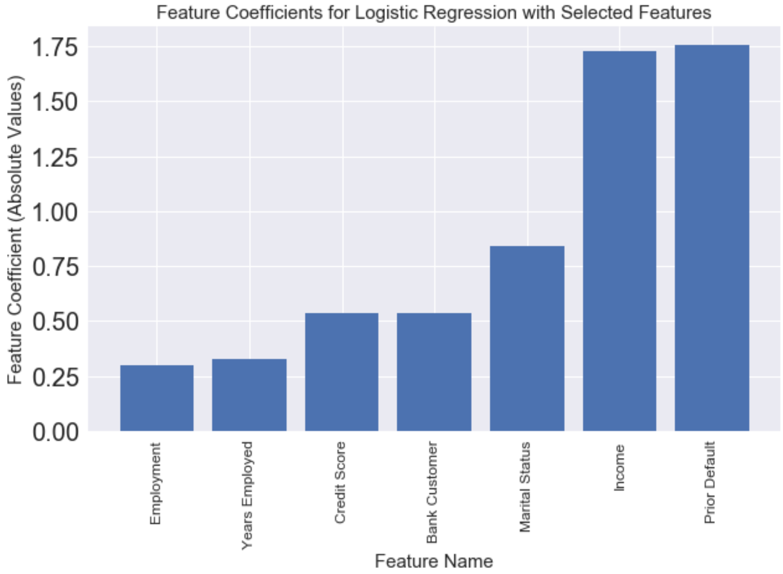
✓ Predict Approval
✖ Predict Denial

Applicant has a ‘**Prior Default**’?

Yes ✖

No ✓

85% Prediction Accuracy for very simple **Decision Tree** using ‘**Prior Default**’ feature!



Feature Coefficients for Logistic Regression with Selected Features

Limitations

Some limitations of this work include:

- The findings in this project (e.g. that ‘**Prior Default**’ can be used as a strong predictor for credit approval or denial) rely entirely on the dataset used, which comes from bank customers of one specific bank in the 1980’s. A different bank may have very different criteria for a credit approval (i.e. a higher risk credit might ignore an applicants ‘**Prior Default**’ status or ‘**Credit Score**’ for example). These findings only hold true for this particular dataset.
- Some of the findings reported in this project depend on random chance, for example, the resulting tree structure from a **Decision Tree** classifier can vary depending on the random seed used. Another example would be reordering the samples in the dataset (by e.g. shuffling) can sometimes produce slightly different results for the trained models, for example, the optimum value of **k** for the **k-NN** classifier may change. However the overall accuracy of each of the models do not vary significantly, even when the dataset is shuffled.
- This analysis has used tentative feature names in order to more easily identify which specific feature is being discussed. These feature names may not be correct, and the feature may relate to some other detail of a credit approval application. Nonetheless, the important features of the dataset for predicting application approval or denial have been determined.

Conclusions

The ‘**Credit Approval**’ dataset was analysed to determine which application details (dataset features) are the most important determiners of credit card application approval or denial:

- Fitting classification models to the numerical features of the dataset indicated that applicant ‘**Credit Score**’, ‘**Years Employed**’, ‘**Income**’ and ‘**Debt**’ were the most important. However, when the categorical variables were also included in the training data, applicant ‘**Prior Default**’ status was found to be a dominant predictor, particularly for the Decision Tree model.

k-Nearest Neighbors, **Decision Tree** and **Logistic Regression** classifiers were trained on the dataset:

- **Using only the numeric features, all three models achieved accuracy (76-78%)** above the baseline accuracy (55%), however while these models had high specificity (91-96%), they had lower sensitivity (52-60%).
- **The same models trained on selected high-importance numeric and categorical variables all scored even higher accuracy (85%)**, with a high sensitivity (86-90%), however a slightly reduced specificity (81-84%).
- The exact model the bank may wish to use to automatically predict approvals would depend on their tolerance of false positives or false negatives i.e. if the bank’s criteria was solely to minimise ‘bad’ customers being approved for credit (false positives), they would likely prefer the k-NN model using only numeric features as it obtained the highest specificity (96%).

Acknowledgements

‘Credit Approval’ Dataset courtesy of UCL Machine Learning Repository:
https://archive.ics.uci.edu/ml/datasets/Credit+Approval

Suggested names for dataset features from ‘Analysis of Credit Approval Data’ by R. Kuhn:
http://rstudio-pubs-static.s3.amazonaws.com/73039_9946de135c0a49daa7a0a9eda4a67a72.html

Credit Card Image courtesy of wikipedia:
https://upload.wikimedia.org/wikipedia/commons/4/4f/Credit-cards.jpg

Thanks to my wife Meg for listening to me talk through my ideas for this project!

References

Original Dataset Submission and Paper by J.R. Quinlan:
Quinlan. "Simplifying decision trees", Int J Man-Machine Studies 27, Dec 1987, pp. 221-234.

‘Analysis of Credit Approval Data’ by R. Kuhn:
http://rstudio-pubs-static.s3.amazonaws.com/73039_9946de135c0a49daa7a0a9eda4a67a72.html

Credit card ownership statistics from creditcards.com:
https://www.creditcards.com/credit-card-news/ownership-statistics/

Matplotlib / Pandas / Seaborn / Scikit-Learn Documentation

Stack Overflow

Week 9 Project-Complete

August 10, 2020

1 Week 9 Project: UCI MLR Credit Approval Data Set

1.1 Paul L. Coster

For the final project of the UCSD Python for Data Science course, I have selected the UCI Machine Learning Repository Credit Approval Data Set. This dataset holds credit card application information, where each sample (row) is an application for a credit card, each sample is labeled with whether the application was approved or not.

The features of the dataset have been changed to meaningless symbols in order to protect the confidentiality of the data. There are 16 different features (columns 0 to 15), of which 6 are numerical values, with the rest being categorical (and column 15 being the approved/not approved label for each sample. There are also 37 rows (5% of the samples) with one or more missing values.

Overall there are 690 samples in the dataset, 307 where the application was approved (44.5%) and 383 where the application was denied (55.5%).

The dataset can be downloaded from the UCI Machine Learning Repository, at this link: https://archive.ics.uci.edu/ml/datasets/Credit+Approval

2 Exploring the Dataset:

The credit approval dataset consists of a single csv file (crx.data), which as described above contains 690 samples of 16 features. While the features are not identified and categorical varibles have been converted to meaningless symbols, analysis of the dataset by others and comparison to the information required for credit card applications has led to some suggested identifications:

Column No.	Id
0	Sex
1	Age
2	Debt
3	Marital Status
4	Bank Customer
5	Education Level
6	Ethnicity
7	Years Employed
8	Prior Default on Credit
9	Emploment Status
10	Credit Score

Column No.	Id
11	Drivers License
12	Citizenship Status
13	ZipCode
14	Income
15	Credit Approval (Sample Label)

See this analysis of the dataset in R for more information about these labels.

While it is not guaranteed that these identifications are correct (and even with them the meaning of the categorical variables in each feature is not clear), they will be useful going forward to give more unique names to each feature column.

It should also be noted that the first usage of this dataset appears to be from 1987 in a paper by J.R. Quinlan (Quinlan. "Simplifying decision trees", Int J Man-Machine Studies 27, Dec 1987, pp. 221-234.). In this paper the dataset is described as:

> Assessing consumer credit applications (Credit): The data for this domain were provided by a large bank. Each case concerns an application for credit card facilities described by 9 discrete and 6 continuous attributes, with two decision classes.

Lets explore the dataset a bit, clean it and get it ready for analysis and ML:

[1]:
```
# Import required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
plt.style.use('seaborn')

# Load in the dataset from the provided csv file, with infered column names:
columns = ['Sex', 'Age', 'Debt', 'Marital Status', 'Bank Customer', 'Education␣
 ↪Level', 'Ethnicity', 'Years Employed', 'Prior Default', 'Employment',␣
 ↪'Credit Score', 'Drivers License', 'Citizenship', 'ZipCode', 'Income',␣
 ↪'Approval']
raw_data = pd.read_csv('crx.data', names=columns)
print(raw_data.shape)
raw_data.head(10)
```

```
(690, 16)
```

[1]:
```
  Sex    Age    Debt Marital Status Bank Customer Education Level Ethnicity  \
0   b  30.83   0.000              u             g               w         v
1   a  58.67   4.460              u             g               q         h
2   a  24.50   0.500              u             g               q         h
3   b  27.83   1.540              u             g               w         v
4   b  20.17   5.625              u             g               w         v
```

```
5   b  32.08   4.000              u             g               m         v
6   b  33.17   1.040              u             g               r         h
7   a  22.92  11.585              u             g              cc         v
8   b  54.42   0.500              y             p               k         h
9   b  42.50   4.915              y             p               w         v

   Years Employed Prior Default Employment  Credit Score Drivers License  \
0           1.250             t          t             1               f
1           3.040             t          t             6               f
2           1.500             t          f             0               f
3           3.750             t          t             5               t
4           1.710             t          f             0               f
5           2.500             t          f             0               t
6           6.500             t          f             0               t
7           0.040             t          f             0               f
8           3.960             t          f             0               f
9           3.165             t          f             0               t

  Citizenship ZipCode  Income Approval
0           g   00202       0        +
1           g   00043     560        +
2           g   00280     824        +
3           g   00100       3        +
4           s   00120       0        +
5           g   00360       0        +
6           g   00164   31285        +
7           g   00080    1349        +
8           g   00180     314        +
9           g   00052    1442        +
```

We can see the shape of the data above matches the description of the dataset give by UCI - so everything has been read in correctly! Lets just check that the datatypes have been interpreted correctly, it looks like Age, Debt, Years Employed, Credit Score, Income (and possibly Zipcode, although categorical would usually be appropriate here the zip codes have been changed into meaningless numerical values) should be numerical variables, and the rest categorical:

```
[2]: raw_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 690 entries, 0 to 689
Data columns (total 16 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Sex             690 non-null    object
 1   Age             690 non-null    object
 2   Debt            690 non-null    float64
 3   Marital Status  690 non-null    object
 4   Bank Customer   690 non-null    object
 5   Education Level  690 non-null    object
 6   Ethnicity        690 non-null    object
 7   Years Employed   690 non-null    float64
 8   Prior Default    690 non-null    object
 9   Employment       690 non-null    object
 10  Credit Score     690 non-null    int64
 11  Drivers License  690 non-null    object
 12  Citizenship      690 non-null    object
 13  ZipCode          690 non-null    object
 14  Income           690 non-null    int64
 15  Approval         690 non-null    object
dtypes: float64(2), int64(2), object(12)
memory usage: 86.4+ KB
```

It looks like the Age and Zip columns have not been read in as numerical data, but rather as categorical (object). This is likely because this column contains a missing value (represented in this dataset as '?'). Lets convert all the '?' values in the dataset to the numpy 'nan' value. Then the 'Age' and 'ZipCode' columns can be converted to numerical data types.

[3]:
```
raw_data = raw_data.replace('?', np.nan)
raw_data['Age'] = raw_data['Age'].astype('float64')
raw_data['ZipCode'] = raw_data['ZipCode'].astype('float64')
raw_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 690 entries, 0 to 689
Data columns (total 16 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   Sex              678 non-null    object
 1   Age              678 non-null    float64
 2   Debt             690 non-null    float64
 3   Marital Status   684 non-null    object
 4   Bank Customer    684 non-null    object
 5   Education Level  681 non-null    object
 6   Ethnicity        681 non-null    object
 7   Years Employed   690 non-null    float64
 8   Prior Default    690 non-null    object
 9   Employment       690 non-null    object
 10  Credit Score     690 non-null    int64
 11  Drivers License  690 non-null    object
 12  Citizenship      690 non-null    object
 13  ZipCode          677 non-null    float64
 14  Income           690 non-null    int64
 15  Approval         690 non-null    object
dtypes: float64(4), int64(2), object(10)
memory usage: 86.4+ KB
```

Now that we have made these columns numeric, lets look at some descriptive statistics for the numerical columns of the dataset:

[4]:
```
raw_data.describe()
```

[4]:
```
              Age        Debt  Years Employed  Credit Score      ZipCode  \
count  678.000000  690.000000      690.000000     690.00000   677.000000   
mean    31.568171    4.758725        2.223406       2.40000   184.014771   
std     11.957862    4.978163        3.346513       4.86294   173.806768   
min     13.750000    0.000000        0.000000       0.00000     0.000000   
25%     22.602500    1.000000        0.165000       0.00000    75.000000   
50%     28.460000    2.750000        1.000000       0.00000   160.000000   
75%     38.230000    7.207500        2.625000       3.00000   276.000000   
max     80.250000   28.000000       28.500000      67.00000  2000.000000   

              Income  
count     690.000000  
mean     1017.385507  
std      5210.102598  
min         0.000000  
25%         0.000000  
50%         5.000000  
75%       395.500000  
max    100000.000000  
```

We can see from the table above that the value of 'Debt' varies from 0 to 28 with a mean of 4.76, 'Years employed' from 0 to 28.5 with a mean of 2.2, 'Credit Score' from 0 to 67 with a mean of 2.4 and 'Income' from 0 to 100000 with a mean of around 1000.

The average applicant age is 31.6, with the minimum value being 13.75 and maximum being 80.25. Although these look like ages in years it is not clear if that is the case as in general a person is required to be 18 or older to apply for their own credit card (e.g. in the USA), however younger people can get credit cards authorised on a parents account (and the rules may have been different when and where this dataset was collected).

Lets have a quick look at the applications by those with an age score less than 18:

[5]:
```
print(raw_data[raw_data['Age'] < 18]['Approval'].value_counts())
raw_data[raw_data['Age'] < 18].sort_values('Age')
```

```
-    28
+     7
Name: Approval, dtype: int64
```

[5]:
```
    Sex    Age    Debt Marital Status Bank Customer Education Level Ethnicity  \
510   b  13.75  4.000              y             p               w         v   
390   b  15.17  7.000              u             g               e         v   
658   a  15.75  0.375              u             g               c         v   
25    a  15.83  0.585              u             g               c         h   
651   a  15.83  7.625              u             g               q         v   
```

```
303   a  15.92   2.875              u             g               q         v
355   a  16.00   0.165              u             g              aa         v
408   b  16.00   3.125              u             g               w         v
294   a  16.08   0.335              u             g              ff        ff
517   b  16.08   0.750              u             g               c         v
606   b  16.17   0.040              u             g               c         v
314   b  16.25   0.000              y             p              aa         v
279   b  16.25   0.835              u             g               m         v
298   b  16.33   2.750              u             g              aa         v
312   a  16.33   0.210              u             g              aa         v
469   b  16.33   4.085              u             g               i         h
446   b  16.50   0.125              u             g               c         v
283   a  16.50   1.250              u             g               q         v
363   b  16.92   0.335              y             p               k         v
480   a  16.92   0.500              u             g               i         v
682   b  17.08   3.290              u             g               i         v
409   b  17.08   0.250              u             g               q         v
265   b  17.08   0.085              y             p               c         v
443   b  17.25   3.000              u             g               k         v
482   a  17.33   9.500              u             g              aa         v
472   b  17.42   6.500              u             g               i         v
317   b  17.50  22.000              l            gg              ff         o
285   b  17.58  10.000              u             g               w         h
222   a  17.58   9.000              u             g              aa         v
282   b  17.67   4.460              u             g               c         v
444   a  17.67   0.000              y             p               j        ff
252   a  17.83  11.000              u             g               x         h
667   a  17.92   0.540              u             g               c         v
678   a  17.92  10.210              u             g              ff        ff
688   b  17.92   0.205              u             g              aa         v

     Years Employed Prior Default Employment  Credit Score Drivers License  \
510           1.750             t          t             2               t
390           1.000             f          f             0               f
658           1.000             f          f             0               f
25            1.500             t          t             2               f
651           0.125             f          t             1               t
303           0.085             f          f             0               f
355           1.000             f          t             2               t
408           0.085             f          t             1               f
294           0.000             f          t             1               f
517           1.750             t          t             5               t
606           0.040             f          f             0               f
314           0.250             f          f             0               f
279           0.085             t          f             0               f
298           0.665             f          t             1               f
312           0.125             f          f             0               f
```

```
469           0.415             f          f            0               t
446           0.165             f          f            0               f
283           0.250             f          t            1               f
363           0.290             f          f            0               f
480           0.165             f          t            6               t
682           0.335             f          f            0               t
409           0.335             f          t            4               f
265           0.040             f          f            0               f
443           0.040             f          f            0               t
482           1.750             f          t           10               t
472           0.125             f          f            0               f
317           0.000             f          f            0               t
285           0.165             f          t            1               f
222           1.375             t          f            0               t
282           0.250             f          f            0               f
444           0.000             f          f            0               f
252           1.000             t          t           11               f
667           1.750             f          t            1               t
678           0.000             f          f            0               f
688           0.040             f          f            0               f

    Citizenship  ZipCode  Income Approval
510           g    120.0    1000        +
390           g    600.0       0        -
658           g    120.0      18        -
25            g    100.0       0        +
651           g      0.0     160        -
303           g    120.0       0        -
355           g    320.0       1        -
408           g      0.0       6        -
294           g    160.0     126        -
517           g    352.0     690        +
606           g      0.0       0        +
314           g     60.0       0        -
279           s    200.0       0        -
298           g     80.0      21        -
312           g    200.0       1        -
469           g    120.0       0        -
446           g    132.0       0        -
283           g    108.0      98        -
363           s    200.0       0        -
480           g    240.0      35        -
682           g    140.0       2        -
409           g    160.0       8        -
265           g    140.0     722        -
443           g    160.0      40        -
482           g      0.0      10        -
```

```
472           g     60.0     100        -
317           p    450.0  100000        +
285           g    120.0       1        -
222           g      0.0       0        +
282           s     80.0       0        -
444           g     86.0       0        -
252           g      0.0    3000        +
667           g     80.0       5        -
678           g      0.0      50        -
688           g    280.0     750        -
```

From the details above it seems that of 35 applicants with an ‘Age’ value below 18 in the dataset, only 20% of these were approved (including the applicant with an ‘Age’ of 13.75). The youngest applicant also held a driving license, had been employed for 1.75 units at the time of the application and also had a ‘Prior Default’. I think this shows it will be important not to read too much into the exact values of each variable per applicant, as the exact meaning of each column and the values within it are not certain.

Lets also look at the categorical columns, to get a better understanding of the information they contain. It looks like ‘Prior Default’, ‘Employment’, ‘Drivers License’ are essentially True/False columns, as well as the ‘Approval’ label for each sample:

```
[6]: print(raw_data['Prior Default'].value_counts())
     print(raw_data['Employment'].value_counts())
     print(raw_data['Drivers License'].value_counts())
     print(raw_data['Approval'].value_counts())
```

```
t    361
f    329
Name: Prior Default, dtype: int64
f    395
t    295
Name: Employment, dtype: int64
f    374
t    316
Name: Drivers License, dtype: int64
-    383
+    307
Name: Approval, dtype: int64
```

They only contain categorical values t or f which would appear to make sense for their apparent identifications.

The ‘Approval’ column which labels the samples is marked ‘+’ for successful applications and ‘-’ for unsuccessful applications.

Lets look at the other categorical columns:

[7]:

```
for name in ['Sex', 'Marital Status', 'Bank Customer', 'Education Level',␣
 ↪'Ethnicity', 'Citizenship']:
    print(' ')
    print(raw_data[name].value_counts())
```

```
b    468
a    210
Name: Sex, dtype: int64

u    519
y    163
l      2
Name: Marital Status, dtype: int64

g     519
p     163
gg      2
Name: Bank Customer, dtype: int64

c     137
q      78
w      64
i      59
aa     54
ff     53
k      51
cc     41
x      38
m      38
d      30
e      25
j      10
r       3
Name: Education Level, dtype: int64

v     399
h     138
bb     59
ff     57
z       8
j       8
dd      6
n       4
o       2
Name: Ethnicity, dtype: int64
```

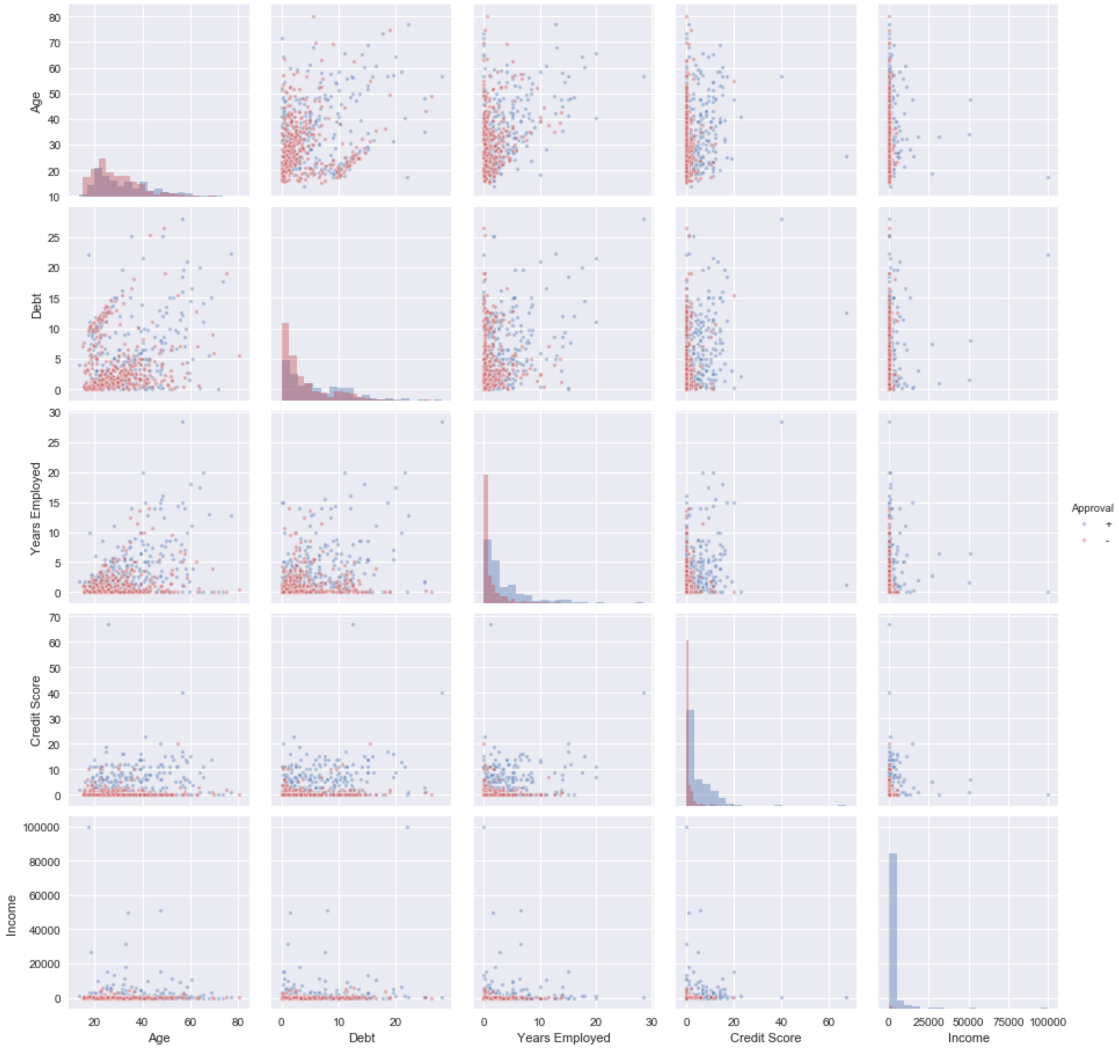
```
g    625
s     57
p      8
Name: Citizenship, dtype: int64
```

We can see that the ‘Sex’ variable only holds values ‘a’ and ‘b’, most likely referring to male or female. It can’t be said for sure which variable refers to which Sex.

There are three categories of both the ‘Marital Status’ and ‘Bank Customer’ variables, 14 categories for Education Level, 9 categories for Ethnicity and 3 categories for citizenship. This agrees with the details given along with the dataset. Great!

Lets try plotting each numerical variable expect zip code against all the other variables to give a quick visual exploration of the data, using seaborn pairplot:

[8]:
```
numeric = ['Age', 'Debt', 'Years Employed', 'Credit Score', 'Income',␣
 ↪'Approval']
sns.pairplot(raw_data[numeric], diag_kind='hist', hue='Approval', markers='.',␣
 ↪palette={'+': '#4c72b0', '-': '#c44e52'}, plot_kws={'alpha': 0.4},␣
 ↪diag_kws={'alpha': 0.4, 'bins': 20})
plt.show()
```



The above plots give a good initial exploration of the numerical part of the dataset. We can see for example that a higher value of credit score appears to be associated with a higher number of approvals, as would be expected (2nd row from the bottom). Many applicants with a low or '0' credit score have applications rejected.

The graph of Years Employed vs Age (Center Row, 1st Column) appears to show that younger applicants with fewer years employed are more likely to have an application rejected, while applicants with a longer employment history are more likely to be approved.

Looking at the Age histogram (top left plot) it can be seen that most rejected applications come from applicants under '40', while approved applications are spread over a wide age range.

It should also be noted at this stage that the histograms for Credit Score and Income denials are not displaying correctly. This appears to be caused by these variables having many repeated '0' values and so the binning for these plots is squashed right into the left hand side of the overall historgram. Creating a custom pairplot grid using seaborn pairgrid could be one solution to get around this in the future.

These plots demonstrate quite well that credit card application approval is a complex procedure which has to take into account many different variables to get the final approval result. Although we can discuss some general trends in where there tend to be more or less approvals, each datapoint is multidimensional, and the approval or denial of the application will depend on many of its variables (each being more or less important).

Clearly manually analyzing each application is a complicated process, and so being able to automate this task to some extent using Machine Learning would be very useful! Lets identify some research questions for this dataset:

3 Research Question(s):

As the data set consists of labelled data, the machine learning problem here is one of classification - can we use a ML model to correctly identify applications as being approved or denied? We have several classification models we can try to use such as linear or logistic classifiers, decision tree classifiers, k-nearest neighbor classifiers etc. Here we will look at:

1. Which application details are the most important determiners of approval or denial in the dataset?
2. Can we build a classification model to predict whether applications will be approved or denied?
    - Do different classification models perform better or worse?
    - Which dataset features are the most important for the classification models?

Before we anwer these questions, lets first clean and prepare the dataset for the various analyses.

4 Data Cleaning and Preparation

Firstly, we know from the initial exploration of the dataset that there are missing values, which were initially marked with '?', and have been converted to numpy NaN values to allow datatype transformations. When dealing with missing values we have several options available such as:

- Remove samples with missing rows entirely
- Impute values for missing values using:
    - Most common value in feature column
    - Average value of feature column (numerical)
    - More complicated imputation strategies such as random regression imputation (using the non-missing variables of the sample to predict the missing variable).

Lets see how many rows are missing data:

[9]:

```
raw_data.isnull().any(axis=1).sum()
```

[9]:

```
37
```

37 of 690 rows are missing data - this is around 5% of the dataset. Since this is only a small portion of the dataset, lets remove rows with missing values from the data:

[10]:

```
cleaned_data = raw_data.dropna()
cleaned_data.reset_index()
cleaned_data.shape
```

[10]:
```
(653, 16)
```

We should check the number of samples we now have of each class, to get an idea of what a baseline accuracy would look like for a naive model with no predictive power:

[11]:
```
print('% Denials and Approvals in Dataset:\n', cleaned_data['Approval'].
 ↪value_counts()/cleaned_data['Approval'].count())
cleaned_data['Approval'].value_counts()
```

```
% Denials and Approvals in Dataset:
 -    0.546708
+    0.453292
Name: Approval, dtype: float64
```

[11]:
```
-    357
+    296
Name: Approval, dtype: int64
```

So the cleaned dataset is made up of around 55% denials and 45% approvals. A naive model that could only pick one class for every sample could therefore achieve a baseline accuracy of 55% by predicting all samples as ‘denied’. Therfore we would like any trained model we create to get an accuracy higher than this.

We have so far dropped the rows with missing values from the dataset. Furthermore for classification tasks, we have some further data preparation to do:

- Convert categorical variables into numeric values.
- Scale the numerical data appropriately (such that e.g. a k-nearest neighbor classifier is not affected by some numerical variables consisting of much wider ranging variables than others).

4.0.1 Converting Categorical Variables to Numeric

To transform categorical variables into numeric values, we can use the sklearn LabelEncoder on the categorical features:

[12]:
```
from sklearn.preprocessing import LabelEncoder

# Copy raw data DF and create dict of encoders for each categorical feature
encoded = cleaned_data.copy()
encoders = {name: LabelEncoder() for name in encoded.columns if encoded[name].
 ↪dtypes == 'object'}

# Iterate through DF categorical columns and convert to numeric values
for feature in encoded.columns:
    if encoded[feature].dtypes == 'object':
        encoded[feature] = encoders[feature].fit_transform(encoded[feature])
```

```
encoded.head(10)
```

```
[12]:    Sex    Age    Debt  Marital Status  Bank Customer  Education Level  \
      0    1  30.83   0.000               1              0               12
      1    0  58.67   4.460               1              0               10
      2    0  24.50   0.500               1              0               10
      3    1  27.83   1.540               1              0               12
      4    1  20.17   5.625               1              0               12
      5    1  32.08   4.000               1              0                9
      6    1  33.17   1.040               1              0               11
      7    0  22.92  11.585               1              0                2
      8    1  54.42   0.500               2              2                8
      9    1  42.50   4.915               2              2               12

         Ethnicity  Years Employed  Prior Default  Employment  Credit Score  \
      0          7           1.250              1           1             1
      1          3           3.040              1           1             6
      2          3           1.500              1           0             0
      3          7           3.750              1           1             5
      4          7           1.710              1           0             0
      5          7           2.500              1           0             0
      6          3           6.500              1           0             0
      7          7           0.040              1           0             0
      8          3           3.960              1           0             0
      9          7           3.165              1           0             0

         Drivers License  Citizenship  ZipCode  Income  Approval
      0                0            0    202.0       0         0
      1                0            0     43.0     560         0
      2                0            0    280.0     824         0
      3                1            0    100.0       3         0
      4                0            2    120.0       0         0
      5                1            0    360.0       0         0
      6                1            0    164.0   31285         0
      7                0            0     80.0    1349         0
      8                0            0    180.0     314         0
      9                1            0     52.0    1442         0
```

Great - looks like we have converted the categorical labels to numerical values successfully. It seems the LabelEncoder has converted approved applications (‘+’) to the value 0 while denied applications (‘-’) to the value 1.

```
[13]: # Demonstrate inverse transform of '0' to '+' using 'Approval' label encoder
      print(encoded['Approval'].value_counts())
      encoders['Approval'].inverse_transform([0])
```

```
1    357
0    296
```

```
Name: Approval, dtype: int64
```

```
[13]: array(['+'], dtype=object)
```

4.0.2 Scaling the features

There are again several possible options when it comes to scaling numerical data for machine learning, these include:

- Min-Max(scaling data into the range 0-1)
- Standardisation (scaling data so the mean is zero and the standard deviation is 1)
- Mean Normalisation (scaling data to have values between -1 and 1 with a mean of 0).
- and many others….

Each scaling technique will transform the data in slightly different ways. For the purpose of this project, lets start by scaling the numerical data using standardisation with sklearn Standard Scaler:

```
[14]: from sklearn.preprocessing import StandardScaler

      # Instantiate Standard Scaler and then transform the encoded dataframe:
      scaler = StandardScaler()
      # Scale only the features and not the target
      X_scaled = scaler.fit_transform(encoded[columns[0:15]])



      # Standard Scaler Returns a numpy array so we have to revert this back to a df␣
       ↪if we want to look at it:
      scaled_data = pd.DataFrame(data=X_scaled, columns=columns[0:15])
      # Add back in Approval column
      scaled_data['Approval'] = encoded['Approval'].values
      scaled_data.head(10)
```

```
[14]:        Sex       Age      Debt  Marital Status  Bank Customer  \
      0  0.671648 -0.056962 -0.961440       -0.536873      -0.554357
      1 -1.488875  2.296536 -0.073565       -0.536873      -0.554357
      2 -1.488875 -0.592078 -0.861903       -0.536873      -0.554357
      3  0.671648 -0.310572 -0.654865       -0.536873      -0.554357
      4  0.671648 -0.958122  0.158358       -0.536873      -0.554357
      5  0.671648  0.048709 -0.165139       -0.536873      -0.554357
      6  0.671648  0.140854 -0.754402       -0.536873      -0.554357
      7 -1.488875 -0.725646  1.344846       -0.536873      -0.554357
      8  0.671648  1.937256 -0.861903        1.800314       1.811624
      9  0.671648  0.929580  0.017014        1.800314       1.811624

         Education Level  Ethnicity  Years Employed  Prior Default  Employment  \
      0         1.450433   0.768656       -0.295171       0.933306    1.129275
      1         0.984553  -0.836244        0.236217       0.933306    1.129275
      2         0.984553  -0.836244       -0.220955       0.933306   -0.885524
      3         1.450433   0.768656        0.446990       0.933306    1.129275
```

```
4         1.450433   0.768656       -0.158613       0.933306   -0.885524
5         0.751614   0.768656        0.075910       0.933306   -0.885524
6         1.217493  -0.836244        1.263368       0.933306   -0.885524
7        -0.878964   0.768656       -0.654378       0.933306   -0.885524
8         0.518674  -0.836244        0.509332       0.933306   -0.885524
9         1.450433   0.768656        0.273325       0.933306   -0.885524

   Credit Score  Drivers License  Citizenship   ZipCode    Income  Approval
0     -0.302596        -0.927577    -0.301777  0.128682 -0.193125         0
1      0.704516        -0.927577    -0.301777 -0.816802 -0.086443         0
2     -0.504019        -0.927577    -0.301777  0.592504 -0.036150         0
3      0.503093         1.078078    -0.301777 -0.477855 -0.192553         0
4     -0.504019        -0.927577     3.347492 -0.358926 -0.193125         0
5     -0.504019         1.078078    -0.301777  1.068219 -0.193125         0
6     -0.504019         1.078078    -0.301777 -0.097283  5.766769         0
7     -0.504019        -0.927577    -0.301777 -0.596784  0.063864         0
8     -0.504019        -0.927577    -0.301777 -0.002140 -0.133307         0
9     -0.504019         1.078078    -0.301777 -0.763284  0.081581         0
```

The values have now been scaled! The should now all have a mean of 0 and a standard deviation of 1:

[15]:
```
scaled_data.describe()
```

[15]:
```
                Sex           Age          Debt  Marital Status  \
count  6.530000e+02  6.530000e+02  6.530000e+02    6.530000e+02
mean  -1.020113e-16 -2.796810e-16 -9.648569e-17   -3.186153e-16
std    1.000767e+00  1.000767e+00  1.000767e+00    1.000767e+00
min   -1.488875e+00 -1.500847e+00 -9.614404e-01   -2.874059e+00
25%   -1.488875e+00 -7.543886e-01 -7.544021e-01   -5.368728e-01
50%    6.716481e-01 -2.606950e-01 -3.970621e-01   -5.368728e-01
75%    6.716481e-01  5.702995e-01  5.316238e-01   -5.368728e-01
max    6.716481e-01  3.824958e+00  4.612666e+00    1.800314e+00

       Bank Customer  Education Level     Ethnicity  Years Employed  \
count   6.530000e+02     6.530000e+02  6.530000e+02    6.530000e+02
mean   -2.506078e-16    -1.190132e-17  1.751194e-17    9.283029e-17
std     1.000767e+00     1.000767e+00  1.000767e+00    1.000767e+00
min    -5.543569e-01    -1.344843e+00 -2.039919e+00   -6.662521e-01
25%    -5.543569e-01    -1.111903e+00 -8.362438e-01   -6.172694e-01
50%    -5.543569e-01     5.279489e-02  7.686561e-01   -3.693874e-01
75%    -5.543569e-01     9.845533e-01  7.686561e-01    1.130177e-01
max     1.811624e+00     1.683372e+00  1.169881e+00    7.794392e+00

       Prior Default    Employment  Credit Score  Drivers License  \
count   6.530000e+02  6.530000e+02  6.530000e+02     6.530000e+02
mean    2.439430e-15 -8.868183e-16  4.495298e-16    -1.207134e-16
```

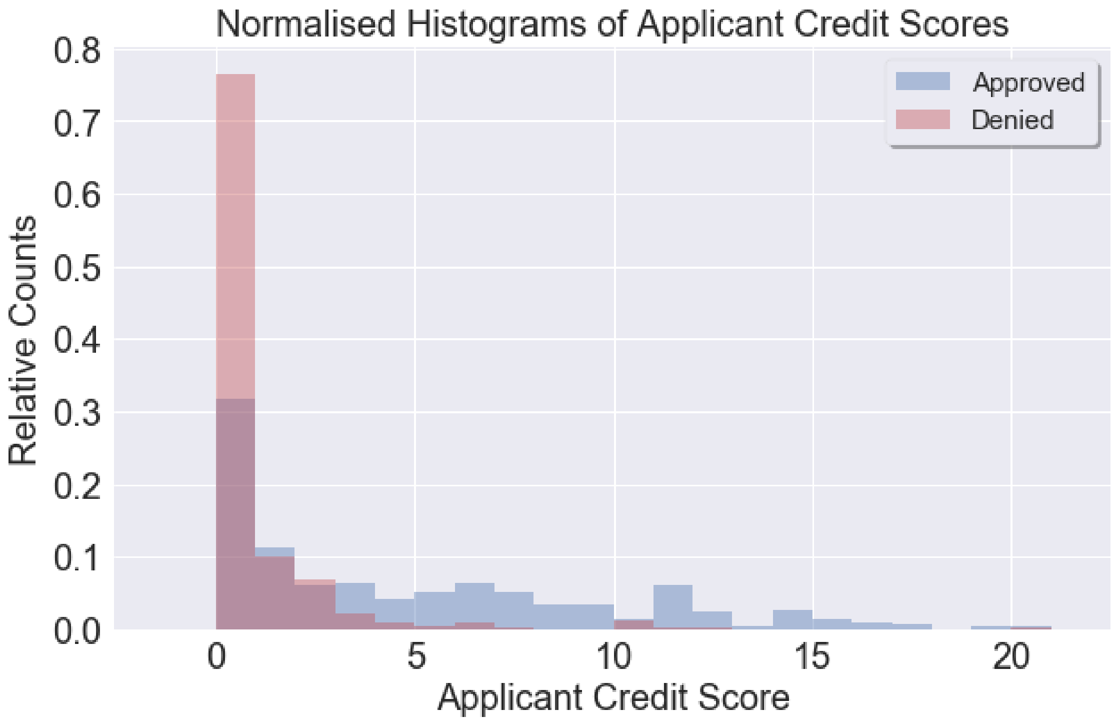
```
std     1.000767e+00  1.000767e+00  1.000767e+00     1.000767e+00
min    -1.071460e+00 -8.855241e-01 -5.040187e-01    -9.275769e-01
25%    -1.071460e+00 -8.855241e-01 -5.040187e-01    -9.275769e-01
50%     9.333060e-01 -8.855241e-01 -5.040187e-01    -9.275769e-01
75%     9.333060e-01  1.129275e+00  1.002485e-01     1.078078e+00
max     9.333060e-01  1.129275e+00  1.299128e+01     1.078078e+00

        Citizenship       ZipCode        Income    Approval
count  6.530000e+02  6.530000e+02  6.530000e+02  653.000000
mean   1.577775e-16  1.173130e-16  1.252614e-16    0.546708
std    1.000767e+00  1.000767e+00  1.000767e+00    0.498195
min   -3.017772e-01 -1.072499e+00 -1.931248e-01    0.000000
25%   -3.017772e-01 -6.384089e-01 -1.931248e-01    0.000000
50%   -3.017772e-01 -1.210688e-01 -1.921723e-01    1.000000
75%   -3.017772e-01  5.449323e-01 -1.169235e-01    1.000000
max    3.347492e+00  1.082038e+01  1.885720e+01    1.000000
```

From the details above we can see that all the columns now have numerical values with a mean very close to 0 and a standard deviation close to 1. Now we can start to analyse the data and carry out some ML tasks on it.

5 1.a. General Trends in Approval/Denial with numerical features:

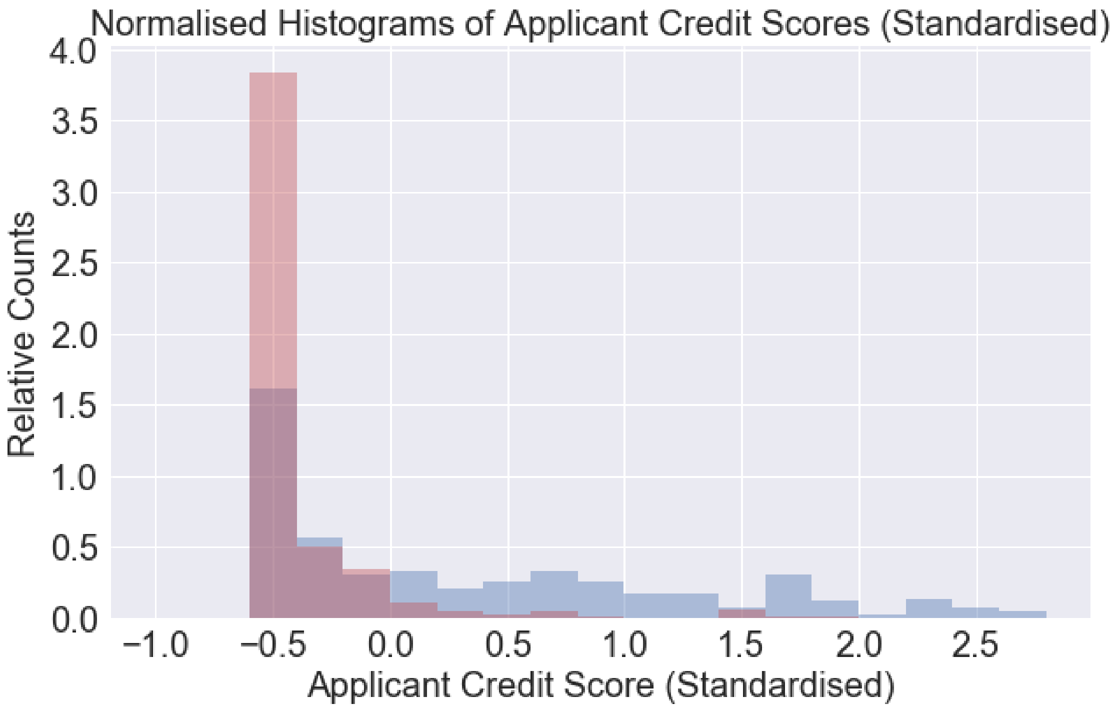
Lets explore some of the plots from the seaborn multiplot in more detail here. Lets first look at a histogram of approved and denied applicants based on credit scores, looking first at the data before we standardised it:

```
[16]: plt.figure(figsize=[10, 6])

plt.xticks(np.arange(0, 30, 5), fontsize=20)
plt.yticks(fontsize=20)
plt.xlim(-2.55, 22.5)
plt.hist(cleaned_data[cleaned_data['Approval'] == '+']['Credit Score'], bins=np.
 ↪arange(0, 31, 1), alpha=0.4, density=True, color='#4c72b0', label='Approved')
plt.hist(cleaned_data[cleaned_data['Approval'] == '-']['Credit Score'], bins=np.
 ↪arange(0, 31, 1), alpha=0.4, density=True,  color='#c44e52', label='Denied')
plt.xlabel('Applicant Credit Score', fontsize=20)
plt.ylabel('Relative Counts', fontsize=20)
plt.title('Normalised Histograms of Applicant Credit Scores', fontsize=20)
plt.legend(fontsize=15, frameon=True, framealpha=1, shadow=True)
plt.show()
```



And the same histogram with the standardised credit scores:

```
[17]: plt.figure(figsize=[10, 6])

plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
#plt.xlim(-2.55, 22.5)
plt.hist(scaled_data[scaled_data['Approval'] == 0]['Credit Score'], alpha=0.4,␣
 ↪bins=np.arange(-1, 3, 0.2), density=True, color='#4c72b0')
plt.hist(scaled_data[scaled_data['Approval'] == 1]['Credit Score'], alpha=0.4,␣
 ↪bins=np.arange(-1, 3, 0.2), density=True,  color='#c44e52')
plt.xlabel('Applicant Credit Score (Standardised)', fontsize=20)
plt.ylabel('Relative Counts', fontsize=20)
plt.title('Normalised Histograms of Applicant Credit Scores (Standardised)',␣
 ↪fontsize=20)
plt.show()
```



The plot above shows that even though the data has been transformed (values standardised), its overall shape is still similar to before the transformation - the data has just been scaled. Lets do a quick calculation to see the percentage of approvals and denials for applicants with the lowest possible credit score:

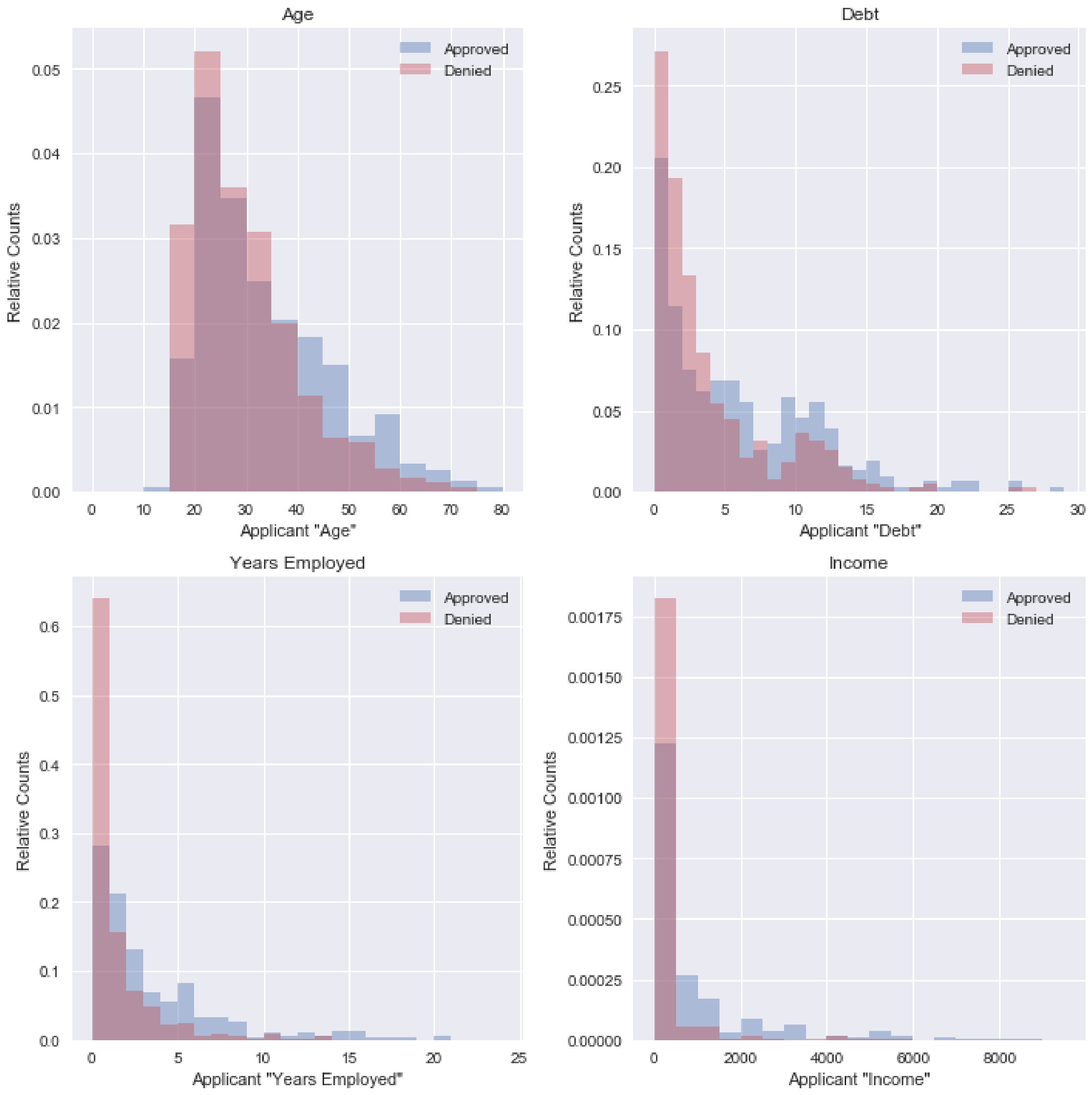
[18]:
```
# Use Boolean indexing to get required subsets of data:

zero_score = cleaned_data['Credit Score'] == 0
denied = cleaned_data['Approval'] == '-'
approved = cleaned_data['Approval'] == '+'
print('0 Credit Score Denied % (Percent of Denials): ', 100 * cleaned_data.
 ↪loc[zero_score & denied].count().max() / cleaned_data.loc[denied].count().
 ↪max())
print('0 Credit Score Denied % (Percent of Applications): ', 100 * cleaned_data.
 ↪loc[zero_score & denied].count().max() / 657)
print('0 Credit Score Approved % (Percent of Approvals): ', 100* cleaned_data.
 ↪loc[zero_score & approved].count().max() / cleaned_data.loc[approved].
 ↪count().max())
print('0 Credit Score Approved % (Percent of Applications): ', 100*␣
 ↪cleaned_data.loc[zero_score & approved].count().max() / 657)
```

```
0 Credit Score Denied % (Percent of Denials):  76.47058823529412
0 Credit Score Denied % (Percent of Applications):  41.55251141552512
0 Credit Score Approved % (Percent of Approvals):  31.41891891891892
0 Credit Score Approved % (Percent of Applications):  14.155251141552512
```

We can see from the histogram above and the calculated percent approvals that the applicant credit score does seem to be quite a useful metric for approvals and denials, with nearly 77% of denials being applicants with 0 credit score, while only around 31% of approvals are for applicants with 0 credit score. Lets try to plot histograms for some of the other numerical variables:

[19]:

```
selected_features = ['Age', 'Debt', 'Years Employed', 'Income']
colors = ['#4c72b0', '#c44e52']
bin_size= [np.arange(0, 85, 5), np.arange(0, 30, 1), np.arange(0, 25, 1), np.
 ↪arange(0, 10000, 500)]
alpha_set = 0.4

# Plot histograms for the selected features in a multiplot:
fig, axs = plt.subplots(2, 2)
fig.set_figheight(10)
fig.set_figwidth(10)
#fig.suptitle('Normalised Histograms of Selected Numerical Features',␣
 ↪fontsize=15)

p = 0
for i in range(2):
    for j in range(2):
        plot = axs[i][j]
        plot.set_title(f'{selected_features[p]}')
        plot.hist(raw_data[raw_data['Approval'] == '+'][selected_features[p]],␣
 ↪density=True, bins=bin_size[p], alpha=alpha_set, color=colors[0],␣
 ↪label='Approved')
        plot.hist(raw_data[raw_data['Approval'] == '-'][selected_features[p]],␣
 ↪density=True, bins=bin_size[p], alpha=alpha_set, color=colors[1],␣
 ↪label='Denied')
        plot.set_ylabel('Relative Counts')
        plot.set_xlabel(f'Applicant "{selected_features[p]}"')
        plot.legend()


        p += 1
plt.tight_layout()
plt.show()
```

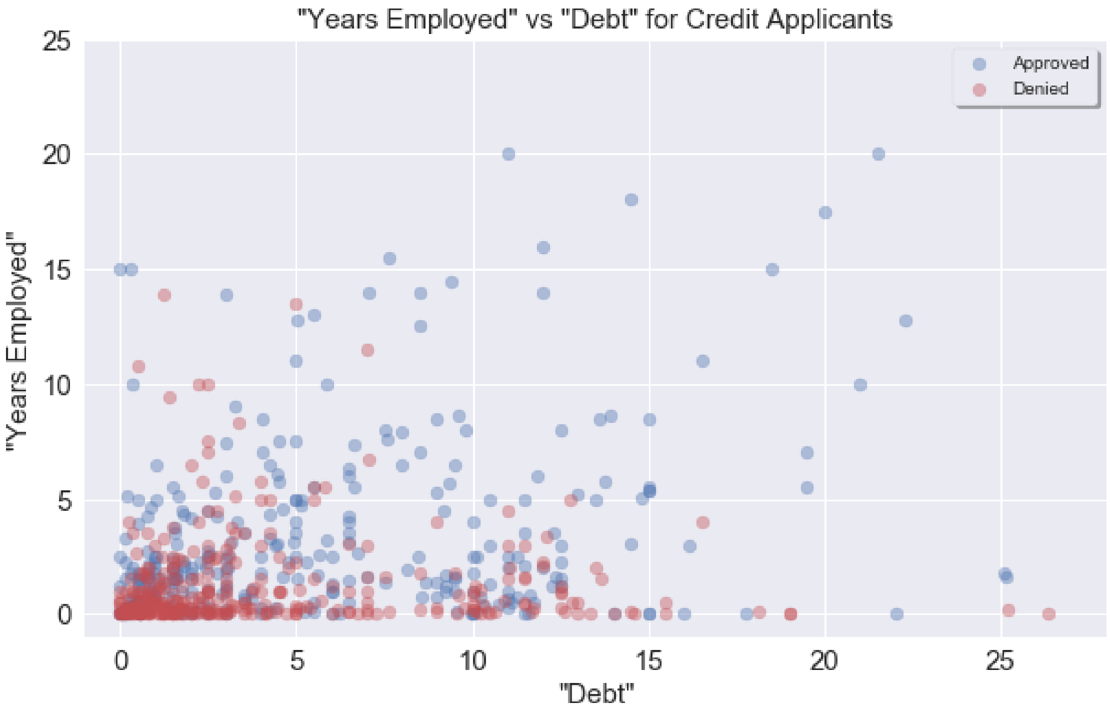


The above histograms look at individual features in terms of applicant denial and approval, based on these we might say:

- Applicants with higher ‘Age’ tend to be approved more (lower ’Age approved less often)
- Applicants with more ‘Years Employed’ tend to be approved more than those with fewer years employed (majority of denied applications have 0 ‘Years Employed’)
- Applicants with a wide range of ‘Debt’ values are both approved and denied. In fact many applicants with 0 ‘Debt’ are denied. This initially does not seem to make sense, however since approval and denial are related to a combination of all the applicants feature variables, perhaps most applicants with 0 debt have a poor credit score/no work history and so have not been approved for any credit/loans previously.
- Applicants with higher ‘Income’ tend to be approved more often, however many applicants with low (0) ‘Income’ are approved as well as denied. Again this clearly shows that credit approval and denial relies on a complex combination of all these variables.

Before building the ML models, lets plot some of the relationships between the numerical variables:

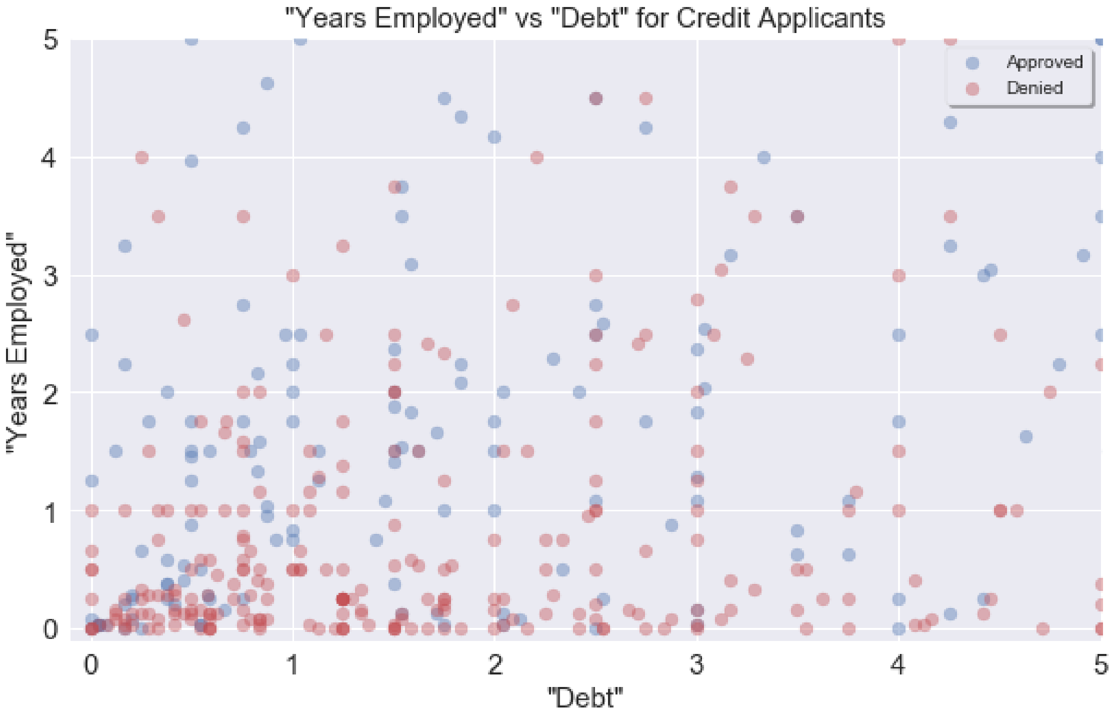
[20]:
```
# Separate data into approved and denied for easy plotting
cleaned_data_approved = cleaned_data[cleaned_data['Approval'] == '+']
cleaned_data_denied = cleaned_data[cleaned_data['Approval'] == '-']

plt.figure(figsize=[10, 6])

plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.ylim(-1, 25)
plt.xlim(-1, 28)

plt.scatter(x=cleaned_data_approved['Debt'], y=cleaned_data_approved['Years␣
 ↪Employed'], c='#4c72b0', alpha=0.4, label='Approved')
plt.scatter(x=cleaned_data_denied['Debt'], y=cleaned_data_denied['Years␣
 ↪Employed'], c='#c44e52', alpha=0.4, label='Denied')
plt.legend(frameon=True, framealpha=1, shadow=True)

plt.title('"Years Employed" vs "Debt" for Credit Applicants', fontsize=15)
plt.ylabel('"Years Employed"', fontsize=15)
plt.xlabel('"Debt"', fontsize=15)

plt.show()
```



The plot above shows from the data that having a higher 'Years Employed' value results in a higher liklihood of approval, regardless of applicant debt. We can zoom in on the region around 0-5 for years employed and debt:

[21]:

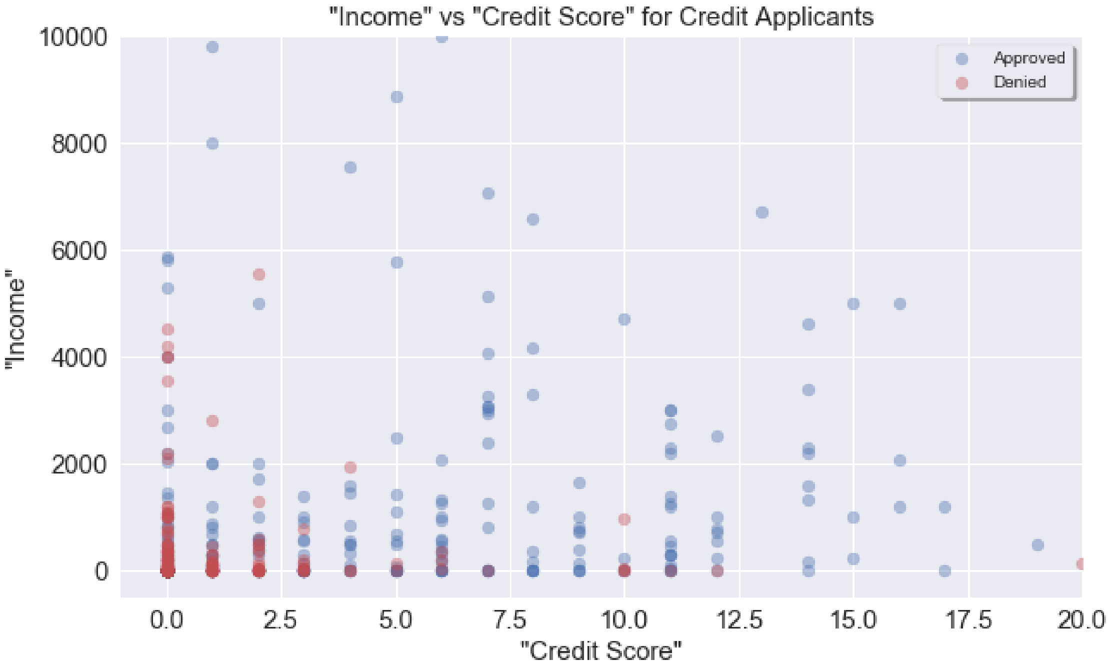
```
plt.figure(figsize=[10, 6])

plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.ylim(-0.1, 5)
plt.xlim(-0.1, 5)

plt.scatter(x=cleaned_data_approved['Debt'], y=cleaned_data_approved['Years␣
 ↪Employed'], c='#4c72b0', alpha=0.4, label='Approved')
plt.scatter(x=cleaned_data_denied['Debt'], y=cleaned_data_denied['Years␣
 ↪Employed'], c='#c44e52', alpha=0.4, label='Denied')
plt.legend(frameon=True, framealpha=1, shadow=True, loc='upper right')

plt.title('"Years Employed" vs "Debt" for Credit Applicants', fontsize=15)
plt.ylabel('"Years Employed"', fontsize=15)
plt.xlabel('"Debt"', fontsize=15)

plt.show()
```



Again the plot is very complex! While there are many denied applications for those with lower

‘Years Employed’, there is a big mixture of approvals and denials in this highlighted region. Finally lets look at ‘credit score’ and ‘income’:

[22]:
```
plt.figure(figsize=[10, 6])

plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.ylim(-500, 10000)
plt.xlim(-1, 20)

plt.scatter(x=cleaned_data_approved['Credit Score'],␣
 ↪y=cleaned_data_approved['Income'], c='#4c72b0', alpha=0.4, label='Approved')
plt.scatter(x=cleaned_data_denied['Credit Score'],␣
 ↪y=cleaned_data_denied['Income'], c='#c44e52', alpha=0.4, label='Denied')
plt.legend(frameon=True, framealpha=1, shadow=True)

plt.title('"Income" vs "Credit Score" for Credit Applicants', fontsize=15)
plt.ylabel('"Income"', fontsize=15)
plt.xlabel('"Credit Score"', fontsize=15)

plt.show()
```



It looks like having low ‘income’ and low ‘credit score’ is likely to have your application denied, which makes sense! We can compare the percentage of approvals and denials with an income less than 2000 and a credit score less than 2.5:

[23]:
```
# Use Boolean indexing to get required subsets of data:
low_score = cleaned_data['Credit Score'] < 2.5
low_income = cleaned_data['Income'] < 2000

print('Low Income and Credit Score Denied %: ', 100 * cleaned_data_denied.
 ↪loc[low_score & low_income].count().max() / 657)
print('Low Income and Credit Score Approved %: ', 100* cleaned_data_approved.
 ↪loc[low_score & low_income].count().max() / 657)
```

```
Low Income and Credit Score Denied %:  49.31506849315068
Low Income and Credit Score Approved %:  18.72146118721461
```

From the above we can see that around 50% of applications with low ‘credit score’ and low ‘income’ are denied, while only 20% are approved. The complicated relationship between all these variables, which results in the application approval or denial, is a perfect machine learning task.

6 1.b. General Trends in Approval/Denial with Catgorical Features:

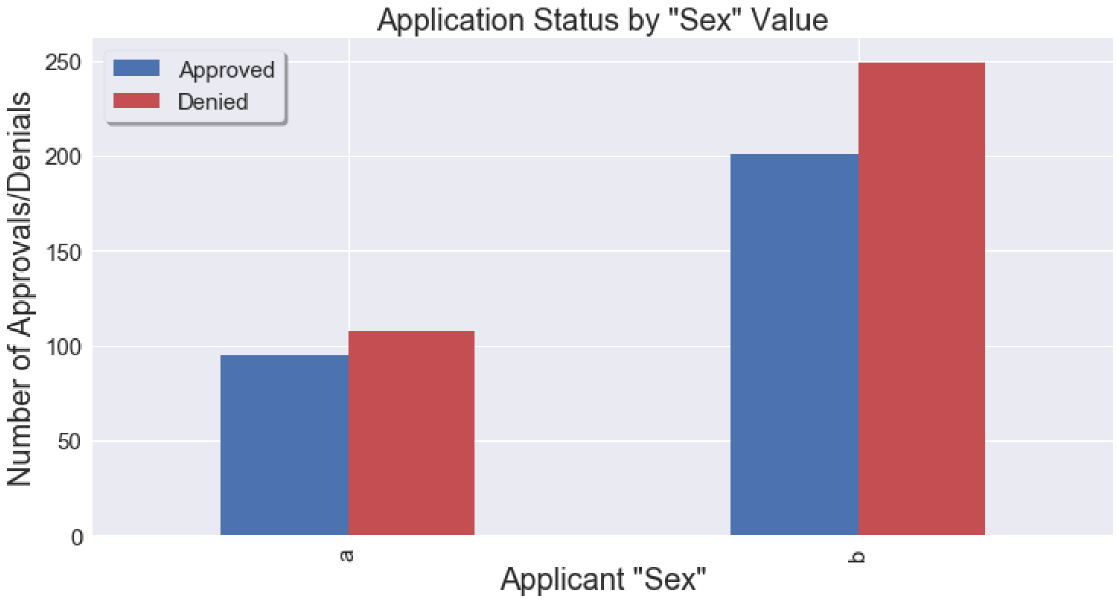
It would be good to look at the approval/denial rates of each categorical feature as well, to see if we can visualise the role they play (if any) in determining if an application will be approved or denied.

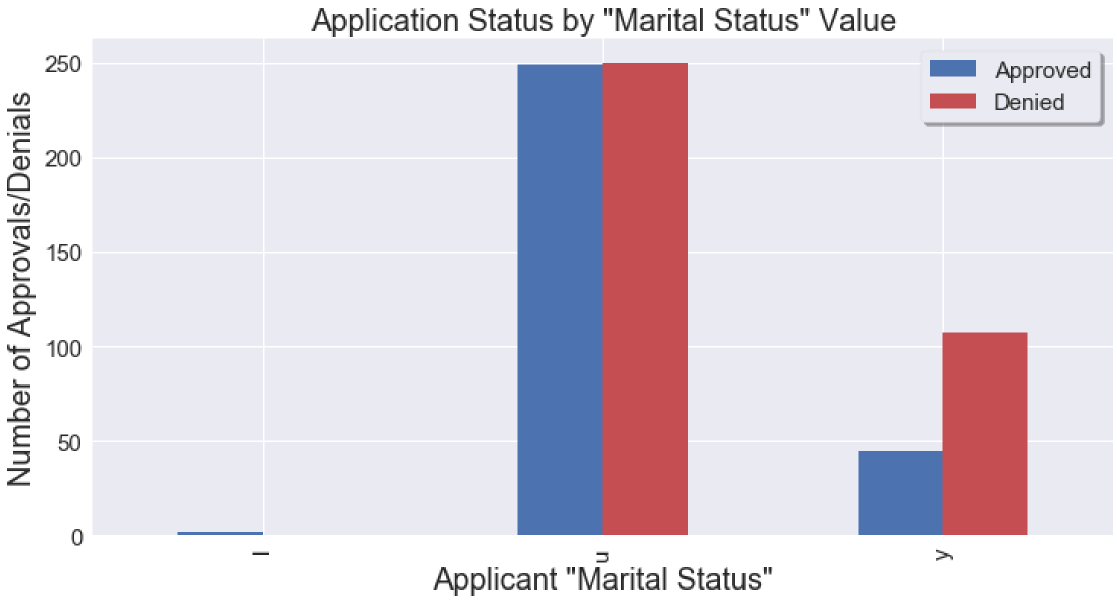
[24]:
```
categorical_features = ['Marital Status', 'Bank Customer', 'Education Level',␣
 ↪'Ethnicity', 'Prior Default', 'Employment', 'Drivers License', 'Citizenship']

cleaned_data.head()
```

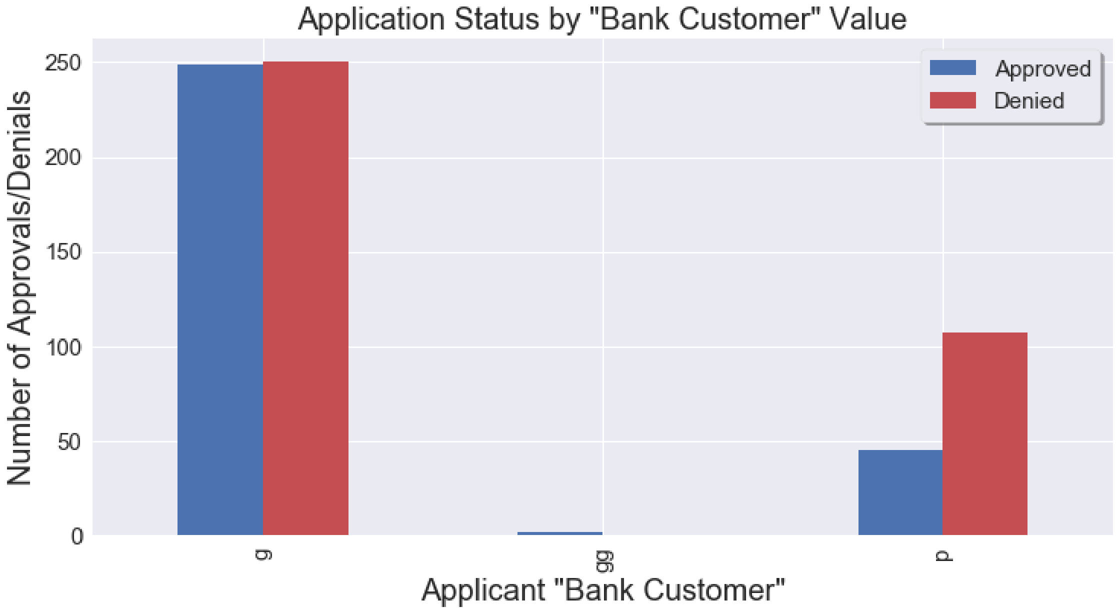
[24]:
```
  Sex    Age   Debt Marital Status Bank Customer Education Level Ethnicity  \
0   b  30.83  0.000              u             g               w         v
1   a  58.67  4.460              u             g               q         h
2   a  24.50  0.500              u             g               q         h
3   b  27.83  1.540              u             g               w         v
4   b  20.17  5.625              u             g               w         v

   Years Employed Prior Default Employment  Credit Score Drivers License  \
0            1.25             t          t             1               f
1            3.04             t          t             6               f
2            1.50             t          f             0               f
3            3.75             t          t             5               t
4            1.71             t          f             0               f

  Citizenship  ZipCode  Income Approval
0           g    202.0       0        +
1           g     43.0     560        +
2           g    280.0     824        +
3           g    100.0       3        +
4            s    120.0        0         +
```

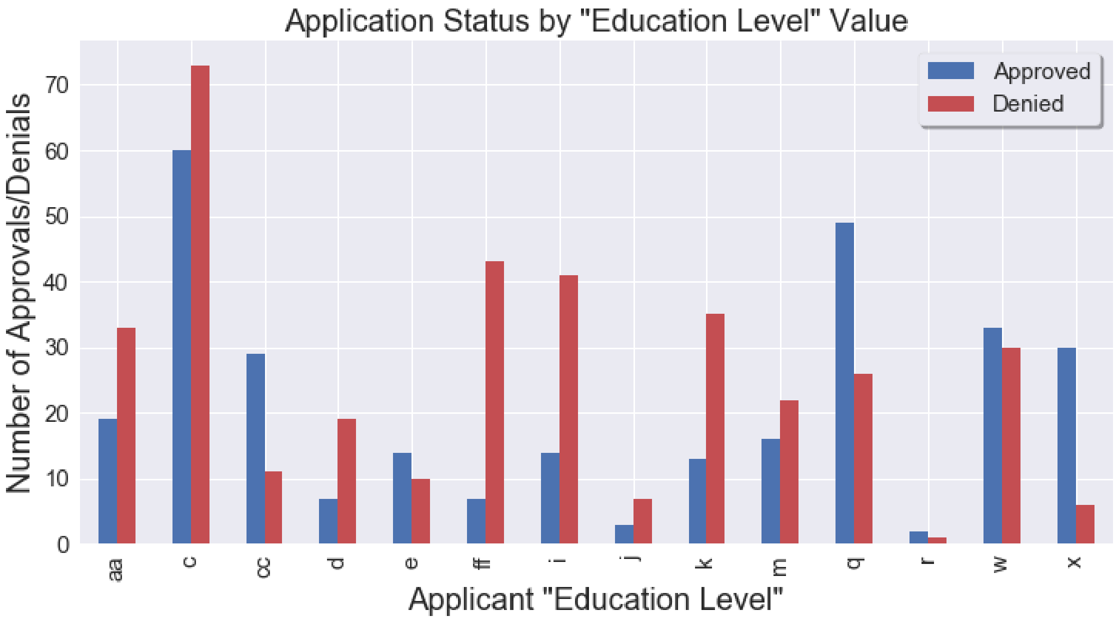
[25]:

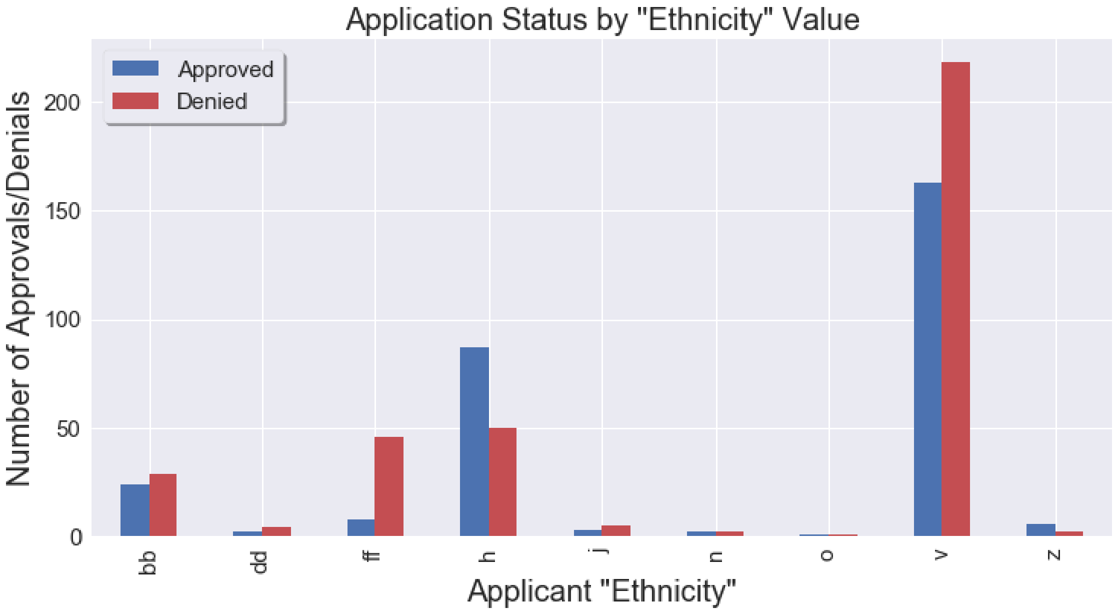
```python
categorical_features = ['Sex', 'Marital Status', 'Bank Customer', 'Education␣
↪Level', 'Ethnicity', 'Prior Default', 'Employment', 'Drivers License',␣
↪'Citizenship']
colors = ['#4c72b0', '#c44e52']

cat_data = cleaned_data.copy()
cat_data = cat_data.replace('+', 'Approved').replace('-', 'Denied').
↪replace('t', 'True').replace('f', 'False')

# Create bar plots for each categorical feature:
for i in range(len(categorical_features)):
        data = cat_data.groupby(categorical_features[i])['Approval'].
↪value_counts().unstack().fillna(0)
        data.index.sortlevel('Approval')
        plot = data.plot(kind='bar', figsize=[12, 6], color=colors)
        plot.set_title(f'Application Status by "{categorical_features[i]}"␣
↪Value', fontsize=20)
        plot.set_ylabel('Number of Approvals/Denials', fontsize=20)
        plot.set_xlabel(f'Applicant "{categorical_features[i]}"', fontsize=20)
        plt.xticks(fontsize=15)
        plt.yticks(fontsize=15)
        plot.legend(fontsize=15, frameon=True, framealpha=1, shadow=True)

plt.tight_layout()
plt.show()
```

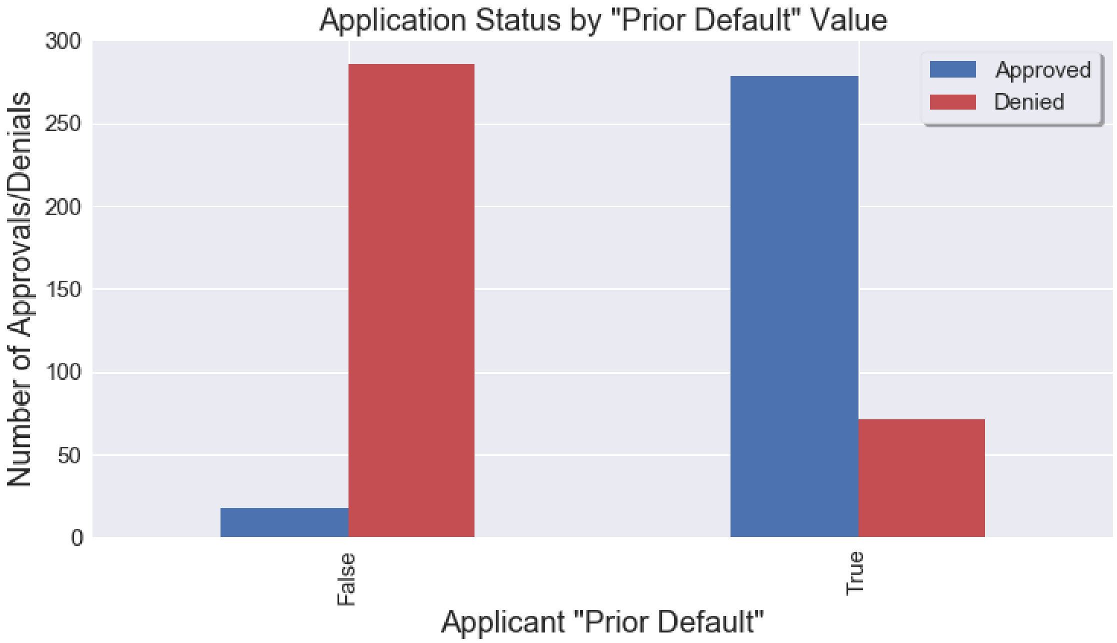


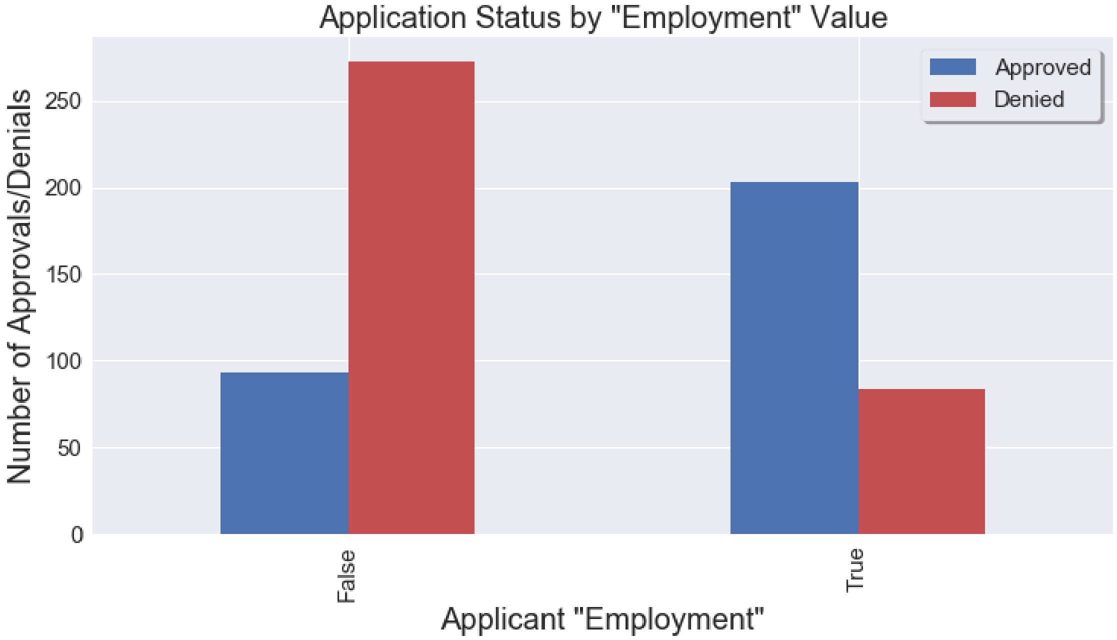

Application Status by "Marital Status" Value
Number of Approvals/Denials
0
50
100
150
200
250
Approved
Denied
l
u
y
Applicant "Marital Status"

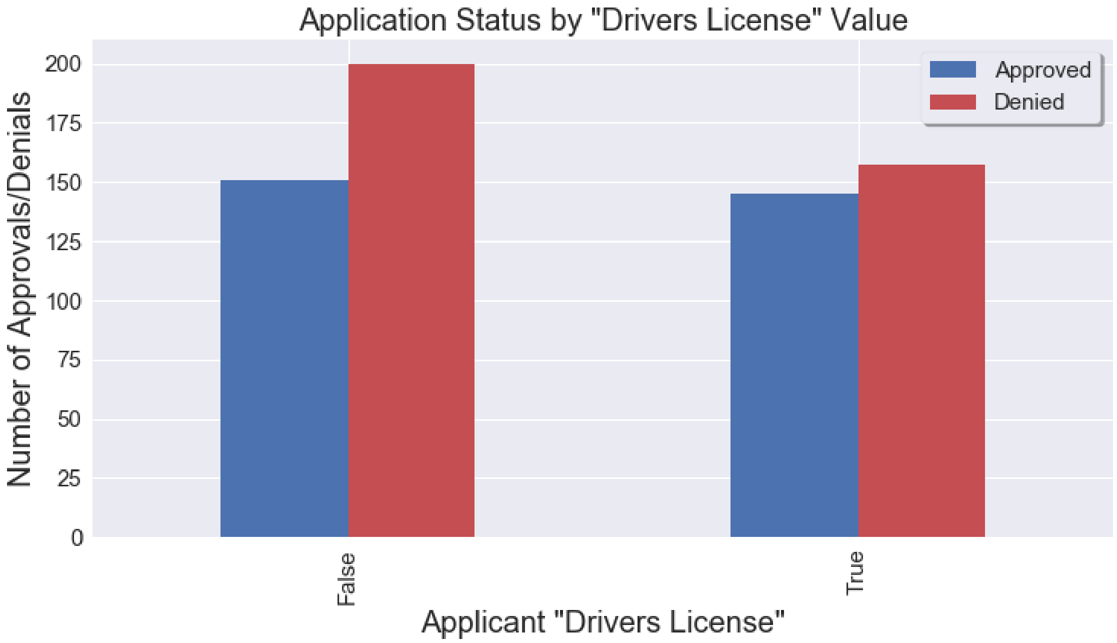

Application Status by "Bank Customer" Value
Number of Approvals/Denials
0
50
100
150
200
250
Approved
Denied
g
gg
p
Applicant "Bank Customer"

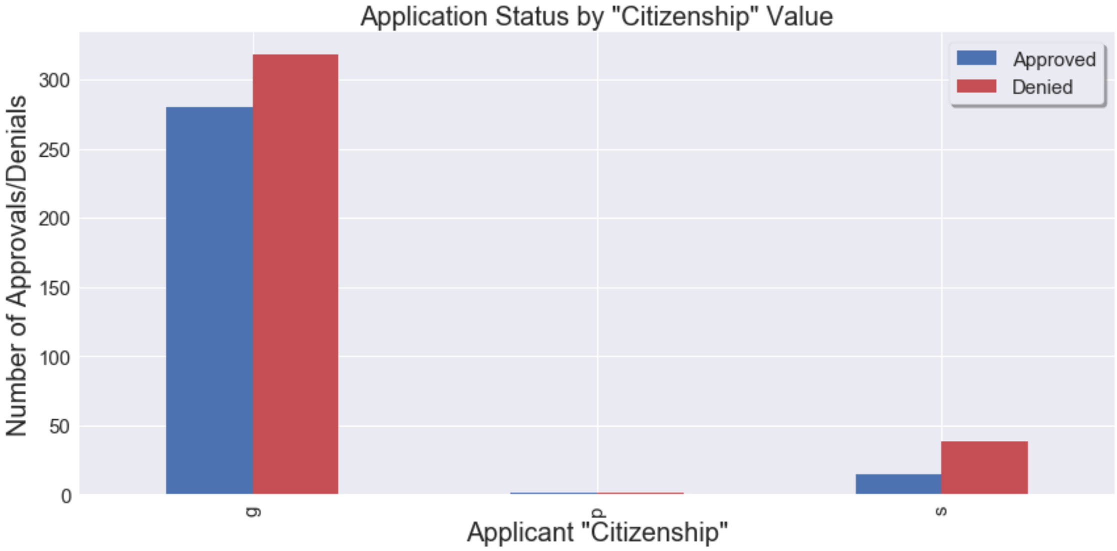

Application Status by "Education Level" Value
Number of Approvals/Denials
0
10
20
30
40
50
60
70
Approved
Denied
aa
c
cc
d
e
ff
i
j
k
m
q
r
w
x
Applicant "Education Level"


Application Status by "Ethnicity" Value
Number of Approvals/Denials
0
50
100
150
200
Approved
Denied
bb
dd
ff
h
j
n
o
v
z
Applicant "Ethnicity"


Application Status by "Prior Default" Value
Number of Approvals/Denials
300
250
200
150
100
50
0
Approved
Denied
False
True
Applicant "Prior Default"


Application Status by "Employment" Value
Number of Approvals/Denials
250
200
150
100
50
0
Approved
Denied
False
True
Applicant "Employment"





Looking through the plots produced above, it looks like the Applicant's 'Prior Default' status as well as their 'employment' status are possibly quite crucial indicators of whether an application will be approved or denied. We can do a quick calculation of the percent of approved and denied applications for each status:

[26]:
```
approved = cat_data['Approval'] == 'Approved'
employed = cat_data['Employment'] == 'True'
defaulted = cat_data['Prior Default'] == 'True'
```

```
print('% of Employed Applicants Approved: ', cat_data.loc[employed & approved].
 ↪count().max() / cat_data.loc[employed].count().max())
print('% of Unemployed Applicants Approved: ', cat_data.loc[~employed &␣
 ↪approved].count().max() / cat_data.loc[~employed].count().max())

print('% of Previously Defaulting Applicants Approved: ', cat_data.
 ↪loc[defaulted & approved].count().max() / cat_data.loc[defaulted].count().
 ↪max())
print('% of Non-Defaulting Applicants Approved: ', cat_data.loc[~defaulted &␣
 ↪approved].count().max() / cat_data.loc[~defaulted].count().max() )
```

```
% of Employed Applicants Approved:  0.7073170731707317
% of Unemployed Applicants Approved:  0.2540983606557377
% of Previously Defaulting Applicants Approved:  0.7965616045845272
% of Non-Defaulting Applicants Approved:  0.05921052631578947
```

70% of employed applicants are approved while only 25% of unemployed applicants area approved. An even bigger contrast can be seen for the 'Prior Default' feature, where applicants with a prior default only have around a 6% Approval rate, while applicants with no prior defaults have around an 80% approval rate. Clearly some rough predictions about a credit application approval could be made based on these features alone, and so they will likely be important features to create a more accuract ML model.

Other categorical features such as 'Sex', 'Citizenship' and 'Drivers License' have approvals closer to 50% for all their values, and so are potentially going to be less useful to the ML model.

Lets build some models, starting with just the numerical features, adding in the categorical features, and then removing any features from the model that are not resulting in increased predictive power.

7 2.a. Classification using Numerical Features

We are going to start by using only the numerical features for the dataset. These are applicant Age, Debt, Years Employed, Credit Score, Income. Zipcode has also been encoded as a numerical column, however we will ignore this column for this analysis - it would seem sensible that applicant Zipcode should not have an effect their credit approval.

(This can be checked by adding Zipcode to the num_features list below - it actually reduced k-NN accuracy, is not used at all by the best DecisionTree classifier and has a small importance (small coefficient) in the Logistic Regression without improving accuracy, so I think it is justified in being ignored!)

Lets compare the effectiveness of different classification models, to see which works best for this particular dataset. In order to be able to compare the effectiveness of the models, we will use crossfold validation (CV) to repeatedly split the dataset into different training and testing sets, and average the overall results of each model across all the train and test sets produced. This allows us to more fairly compare the models against each other, as using something like train_test_split alone will select a random subset of the data each time we run it, and so runs cannot be compared directly to one another.

7.0.1 K-NN Classifier:

Lets start by using the k-NN classifier from sklearn. We can also use Grid Search to vary the value of ‘k’ so that we can determine the best value for ‘k’ in this dataset.

[27]:
```
# Import required tools:
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

num_features = ['Age', 'Debt', 'Years Employed', 'Credit Score', 'Income']

# Set grid parameters
grid_params = {'n_neighbors': [1, 3, 5, 7, 9, 11,13, 15, 17, 19, 21, 23, 25,␣
 ↪27, 29, 31, 33, 35, 37, 39, 41]}

# Create grid search CV
gs = GridSearchCV(KNeighborsClassifier(), grid_params, verbose=1, cv=5, n_jobs=␣
 ↪-1, scoring='accuracy')

# Get np array of numerical features
X_num = scaled_data[num_features].values
y = scaled_data['Approval'].values

# Fit the Grid Search CV
gs_results = gs.fit(X_num, y)
```

```
Fitting 5 folds for each of 21 candidates, totalling 105 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done  82 out of 105 | elapsed:    1.3s remaining:    0.3s
[Parallel(n_jobs=-1)]: Done 105 out of 105 | elapsed:    1.3s finished
```

The fitted gridCV produces a results table that we can put into a Pandas DataFrame to look at the results:

[28]:
```
knn_num_results = pd.DataFrame(gs_results.cv_results_)
knn_num_results.head()
```

[28]:
```
   mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0       0.001488      0.000313         0.007142        0.000972
1       0.001686      0.000673         0.008730        0.001389
2       0.001091      0.000199         0.007638        0.002072
3       0.001290      0.000243         0.008730        0.001678
4       0.001190      0.000243         0.007936        0.001568

  param_n_neighbors               params  split0_test_score  split1_test_score  \
0                 1  {'n_neighbors': 1}           0.664122           0.778626
1                 3  {'n_neighbors': 3}           0.679389           0.763359
```

```
2                 5  {'n_neighbors': 5}          0.656489           0.809160
3                 7  {'n_neighbors': 7}          0.679389           0.816794
4                 9  {'n_neighbors': 9}          0.687023           0.839695

   split2_test_score  split3_test_score  split4_test_score  mean_test_score  \
0           0.603053           0.669231           0.746154         0.692237
1           0.717557           0.676923           0.723077         0.712061
2           0.748092           0.669231           0.730769         0.722748
3           0.755725           0.646154           0.746154         0.728843
4           0.763359           0.661538           0.746154         0.739554

   std_test_score  rank_test_score
0        0.062679               21
1        0.031891               20
2        0.055548               19
3        0.060068               17
4        0.062425               14
```

[29]:

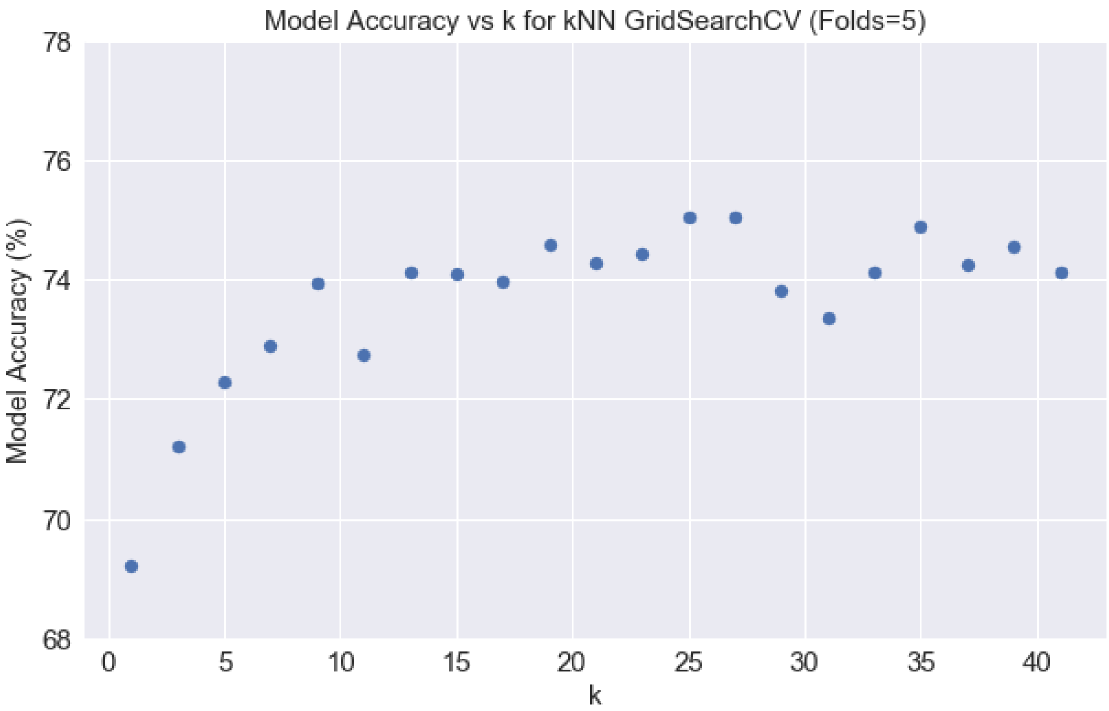
```
print(gs_results.best_score_, gs_results.best_params_)
```

```
0.7503229594832648 {'n_neighbors': 25}
```

We managed to get 75% accuracy just using KNN (with k=25) on the numerical features alone! However we do need to bear in mind that with only two classes to select, and 55% of the samples being the ‘denied’ class, a baseline accuracy (all samples predicted to be denied) would give 55% accuracy. Lets plot the accuracy of each model vs k:

[30]:

```
plt.figure(figsize=[10, 6])

plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.ylim(68, 78)

plt.scatter(x=grid_params['n_neighbors'],␣
 ↪y=knn_num_results['mean_test_score']*100)

plt.title('Model Accuracy vs k for kNN GridSearchCV (Folds=5)', fontsize=15)
plt.ylabel('Model Accuracy (%)', fontsize=15)
plt.xlabel('k', fontsize=15)

plt.show()
```



It looks like the accuracy starts to drop off as we take k >> 25.

7.0.2 Decision Tree Classifier:

Lets look at a different type of classifier, the decision tree. Here we can set hyperparameters such as the max tree depth and the max leaf nodes.

[31]:

```
# Import Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree

tree_params = {'max_depth': [1, 2, 3, 4, 6, 8], 'max_leaf_nodes': [4, 8, 16],␣
 ↪'random_state':[42]}

gs_tree = GridSearchCV(DecisionTreeClassifier(), tree_params, verbose=1, cv=5,␣
 ↪n_jobs=-1, scoring='accuracy')

# Fit the Grid Search CV
gs_tree_results = gs_tree.fit(X_num, y)
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done  90 out of  90 | elapsed:    0.0s finished
```

[32]:

```python
tree_num_results = pd.DataFrame(gs_tree_results.cv_results_)
#tree_num_results.head()
```

[33]:

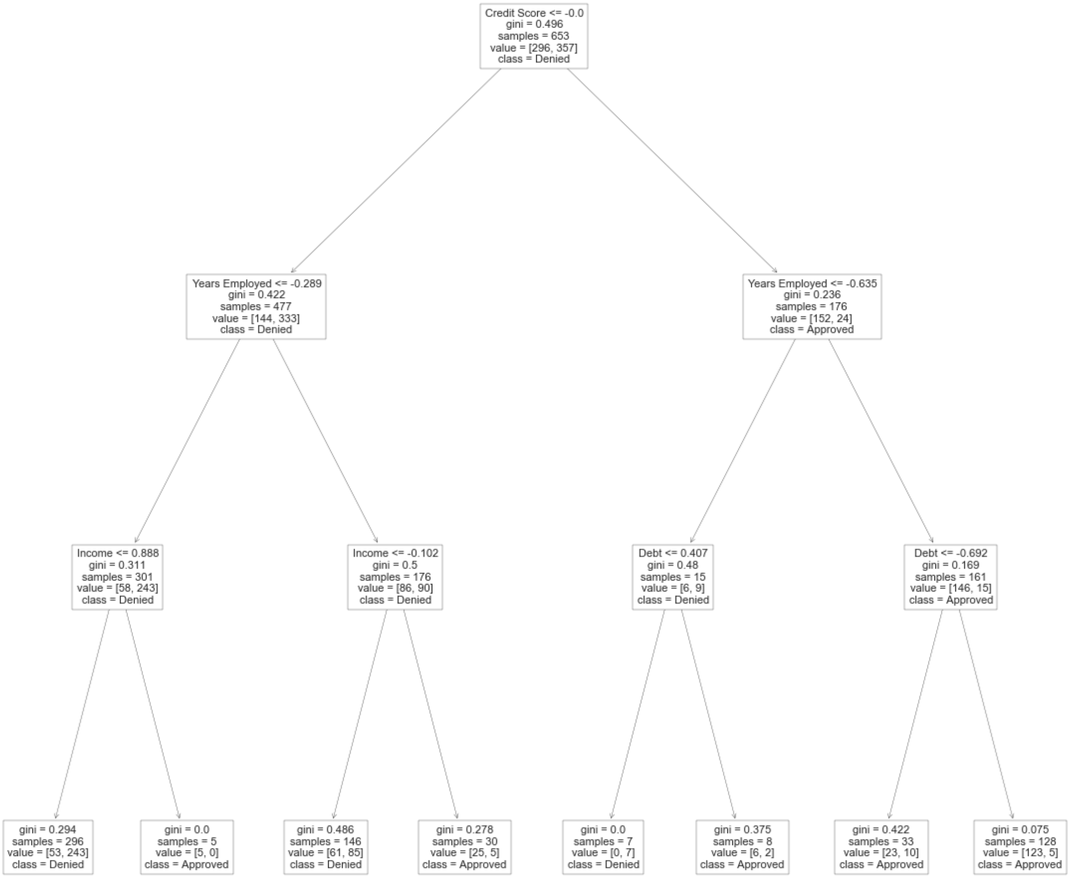
```python
print(gs_tree_results.best_score_, gs_tree_results.best_params_)
```

```
0.7610099823840282 {'max_depth': 3, 'max_leaf_nodes': 8, 'random_state': 42}
```

It looks like the best result for the gridsearch was the tree with a max_depth of only 3! A higher depth tree than this resulting in a worse mean accuracy during the crossfold validation. We can select the best model from the GridSearchCV and look at the tree itself:

[34]:

```python
plt.figure(figsize=[20, 20])
plot_tree(gs_tree_results.best_estimator_, feature_names=num_features,␣
 ↪class_names=['Approved', 'Denied'])
plt.show()
```



From the tree plot above we can see that the first decision in the tree is whether the applicants standardised ‘Credit Score’ is less than or equal to 0 (i.e. if the applicant credit score is less than the mean score).

If this is true (lower than average credit score), it follows the path to the left, otherwise the tree follows the path to the right etc. Interestingly the left hand branch following this decision contains 333 out of 357 denied applications. On the other hand, the left and right branch contain 144 and 152 of the approved applications, respectively. Applicants with credits scores greater than the mean are highly likely to be approved, while those less than the mean are a mixture of ~30% approved and 70% denied.

Interestingly the ‘Age’ feature is not used at all in the decision tree, showing that this feature is largely irrelevant in classifying an application, when using a Decision Tree.

The remaining nodes in the decision tree are trying to separate the mixture of approved/denied applicants from each other.

7.0.3 Logistic Regression:

One other type of ML Classifier we can try on the dataset is logistic regression. Logistic regression can classify our samples and also return a probability associated with that classification. Here the weighted sum of inputs is passed through a sigmoid activation function.

We will again use GridSearchCV on the logistic regression classifier, here we can set hyperparameters such as the tolerance of the stopping criteria, or the number of iterations taken for the solvers to converge.

[35]:
```
# Import Logistic Regression:
from sklearn.linear_model import LogisticRegression

log_params = {'tol': [0.01, 0.001, 0.0001], 'max_iter': [100, 150, 200]}

gs_log = GridSearchCV(LogisticRegression(), log_params, verbose=1, cv=5,␣
 ↪n_jobs=-1, scoring='accuracy')

# Fit the Grid Search CV
gs_log_results = gs_log.fit(X_num, y)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:    0.0s finished
```

[36]:
```
log_num_results = pd.DataFrame(gs_log_results.cv_results_)
#log_num_results.head()
```

[37]:
```
print(gs_log_results.best_score_, gs_log_results.best_params_)
```

```
0.7626189078097476 {'max_iter': 100, 'tol': 0.01}
```

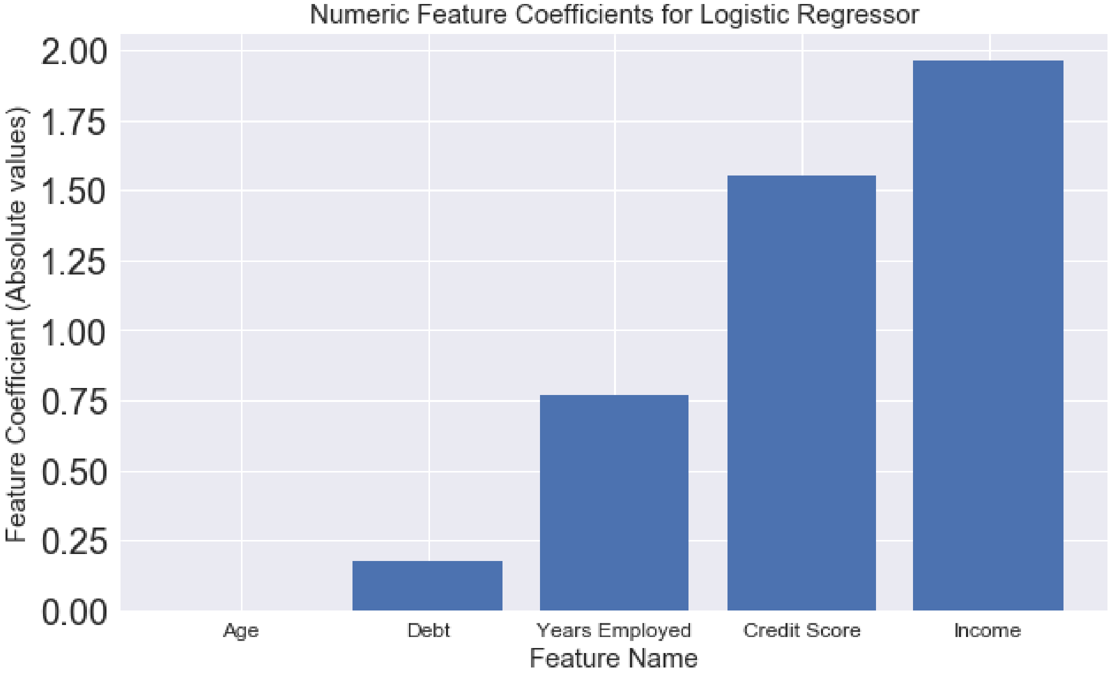
It looks like our best result is with max_iter = 100 and tol = 0.01, with a mean 76% accuracy during the CV. There actually is no difference in accuracy between the different 'tol' and 'max_iter' settings in the grid search, so the grid search across these parameters was not really necessary!

With a fitted logistic regressor, we can use the fitted coefficients for each of the features as a rough measure of the feature importance:

[38]:

```
# Coefficients of the best logreg model (Negate the features because due to the␣
 ↪approval labels they are negative values):
coefs = -gs_log.best_estimator_.coef_[0]
print('Regressor Coefs: ', coefs)
plt.figure(figsize=[10, 6])

plt.yticks(fontsize = 20)
plt.xticks(fontsize = 12)

plt.bar(x=num_features, height=coefs)

plt.title('Numeric Feature Coefficients for Logistic Regressor', fontsize=15)
plt.ylabel('Feature Coefficient (Absolute values)', fontsize=15)
plt.xlabel('Feature Name', fontsize=15)

plt.show()
```

```
Regressor Coefs:  [1.04989475e-03 1.76583047e-01 7.67320597e-01 1.54928553e+00
 1.96175818e+00]
```



As with the Decision Tree, we can see that the Logistic Regressor finds the ‘Age’ Feature to be of no importance! Applicant ‘Debt’ also appears to perhaps be of lower importance while Years Employed, Credit Score and Income are the most important features (largest coefficients) for the Logistic Regressor.

So far we have created models only on the numerical features of the dataset, with the best found models having accuracies of:

- kNN: 75%
- Decision Tree: 76%
- Logistic Regression: 76%

We have seen that of the numerical data, the ‘Age’ feature does not seem to be very important for two of the models made so we could perhaps simplify future models by removing it.

While the GridSearchCV mean accuracy for all the models is quite similar, they could potentially be quite different in terms of true positive / false positive / true negative / false negative rates. Lets take the best models from each classifier, split the data into a stratified train and test set and create confusion matrices for each model:

[39]:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix

# Carry out stratified train test split (keep ratio of approved and denied␣
 ↪similar)
X_num_train, X_num_test, y_num_train, y_num_test = train_test_split(X_num, y,␣
 ↪stratify=y, random_state=42, test_size=0.33)

models = [('k-Nearest Neighbors', KNeighborsClassifier(n_neighbors=25)),␣
 ↪('Decision Tree', DecisionTreeClassifier(max_depth=3, max_leaf_nodes=8,␣
 ↪random_state=42)), ('Logistic Regression', LogisticRegression(max_iter=100,␣
 ↪tol=0.01))]
results = list()

# Iterate through the models:
for name, model in models:

    # Get model accuracy
    model.fit(X_num_train, y_num_train)
    y_pred = model.predict(X_num_test)
    acc = model.score(X_num_test, y_num_test)
    print(f'{name} accuracy: ', acc*100)
    results.append(('Accuracy', name, acc*100))

    # Get confusion matrix
    print(f'{name} confusion matrix:')
    c_m = confusion_matrix(y_num_test, y_pred)
    print(c_m)
    # Get sensitivity and specificity
    tpr = c_m[0][0]/(c_m[0][0] + c_m[0][1])
    tnr = c_m[1][1]/(c_m[1][1] + c_m[1][0])
    print('True Positive Rate: ', tpr*100)
    print('True Negative Rate: ', tnr*100)
    results.append(('Sensitivity', name, tpr*100))
    results.append(('Specificty', name, tnr*100))
    #plot_confusion_matrix(model, X_num_test, y_num_test)

    print('')

print(results)
```

```
k-Nearest Neighbors accuracy:  75.92592592592592
k-Nearest Neighbors confusion matrix:
[[ 51  47]
 [  5 113]]
True Positive Rate:  52.04081632653062
True Negative Rate:  95.76271186440678

Decision Tree accuracy:  77.77777777777779
Decision Tree confusion matrix:
[[ 57  41]
 [  7 111]]
True Positive Rate:  58.16326530612245
True Negative Rate:  94.0677966101695

Logistic Regression accuracy:  76.85185185185185
Logistic Regression confusion matrix:
[[ 59  39]
 [ 11 107]]
True Positive Rate:  60.204081632653065
True Negative Rate:  90.67796610169492

[('Accuracy', 'k-Nearest Neighbors', 75.92592592592592), ('Sensitivity',
'k-Nearest Neighbors', 52.04081632653062), ('Specificty', 'k-Nearest Neighbors',
95.76271186440678), ('Accuracy', 'Decision Tree', 77.77777777777779),
('Sensitivity', 'Decision Tree', 58.16326530612245), ('Specificty', 'Decision
Tree', 94.0677966101695), ('Accuracy', 'Logistic Regression',
76.85185185185185), ('Sensitivity', 'Logistic Regression', 60.204081632653065),
('Specificty', 'Logistic Regression', 90.67796610169492)]
```

The confustion matrices above consist of:

First row - First Element: Approved Applications Correctly Identified (True Positives)

First row - Second Element: Approved Applications Incorrectly Identified as Denied (False Negative)

Second row - Second Element: Denied Applications Incorrectly Identified as Approved (False Positives)

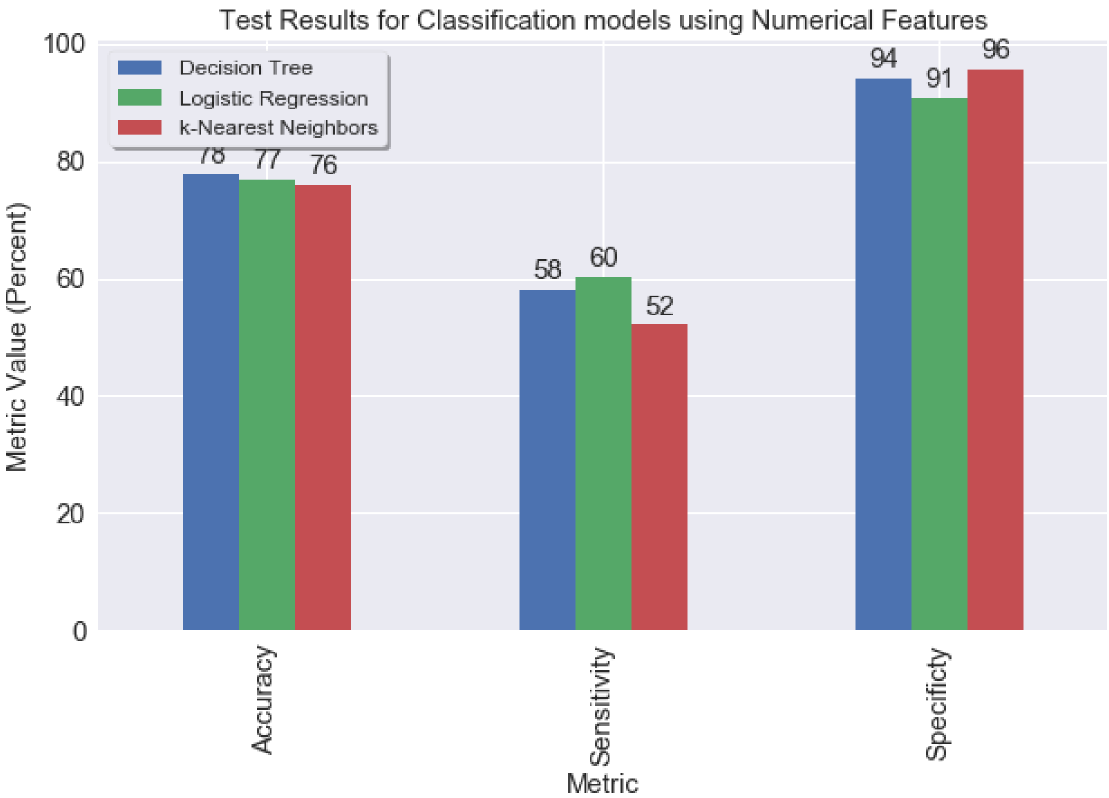
Second row - Second Element: Denied Applications Correctly Identified (True Negatives)

We have also calculate the Sensitivity (True positive rate) and Specificity (True Negative Rate) for each classifier). We can see from these results that while the overall accuracy of the models is similar, the Logreg model has the highest True positive rate while the KNN model has the highest True negative rate. This means that for the KNN model 96% of 'Denied' applications were correctly identified, while for the LogReg model 60% of 'Approved' Applications were correctly identified.

All the models are much better at identifying which applications should be denied (90 to 96% true negative rate) than which ones should be approved (52-60% true positive rate). Depending on the bank's specific criteria, this may make one model preferable over another, i.e. if the bank would prefer to have fewer applications that should be denied as approved (and therfore not give credit to a customer that is more likely perhaps to default) then the KNN model is the best candidate.

Lets create a quick bar plot to summarise the results here:

[40]:
```
num_comp_data = pd.DataFrame(results, columns=['Metric', 'Model', 'Result'])

num_comp_data = num_comp_data.set_index(['Metric', 'Model'])

plot = num_comp_data.unstack().plot(kind='bar', figsize=[10,6], y='Result')

for p in plot.patches:
    plot.annotate(str(int(p.get_height().round(0))), xy=(p.get_x() + p.
 ↪get_width() / 2, p.get_height()),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=15)

plot.set_title(f'Test Results for Classification models using Numerical␣
 ↪Features', fontsize=15)
plot.set_ylabel('Metric Value (Percent)', fontsize=15)
plot.set_xlabel(f'Metric', fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plot.legend(frameon=True, framealpha=1, shadow=True, fontsize=12, loc='upper␣
 ↪left')

plt.show()
```



[41]:

```
#Check that tree trained on 66% dataset is similar to gridsearch tree - they␣
 ↪are mostly similar
#plt.figure(figsize=[20, 20])
#plot_tree(models[1][1], feature_names=num_features, class_names=['Approved',␣
 ↪'Denied'])
#plt.show()
```

Lets add in the categorical features to the models and see if these can improve prediction accuracy:

8 2.b. Classification using numerical and categorical features:

We can now easily repeat what we have done above using the full set of features, which should hopefully improve the accuracy of the models using useful features such as Prior Default status. We will keep the numerical features to Debt, Years Employed, Credit Score and Income as these appear to be the most important numerical features for the models trained so far.

8.0.1 k-NN Classifier:

[42]:

```
num_cat_features = ['Age', 'Sex', 'Debt', 'Marital Status', 'Bank Customer',␣
 ↪'Education Level', 'Ethnicity', 'Years Employed', 'Prior Default',␣
 ↪'Employment', 'Credit Score', 'Drivers License', 'Citizenship', 'Income']

# Get np array of numerical features
X_num_cat = scaled_data[num_cat_features].values
y = scaled_data['Approval'].values

# Fit the Grid Search CV
gs_knn_num_cat_results = gs.fit(X_num_cat, y)
```

```
Fitting 5 folds for each of 21 candidates, totalling 105 fits
```

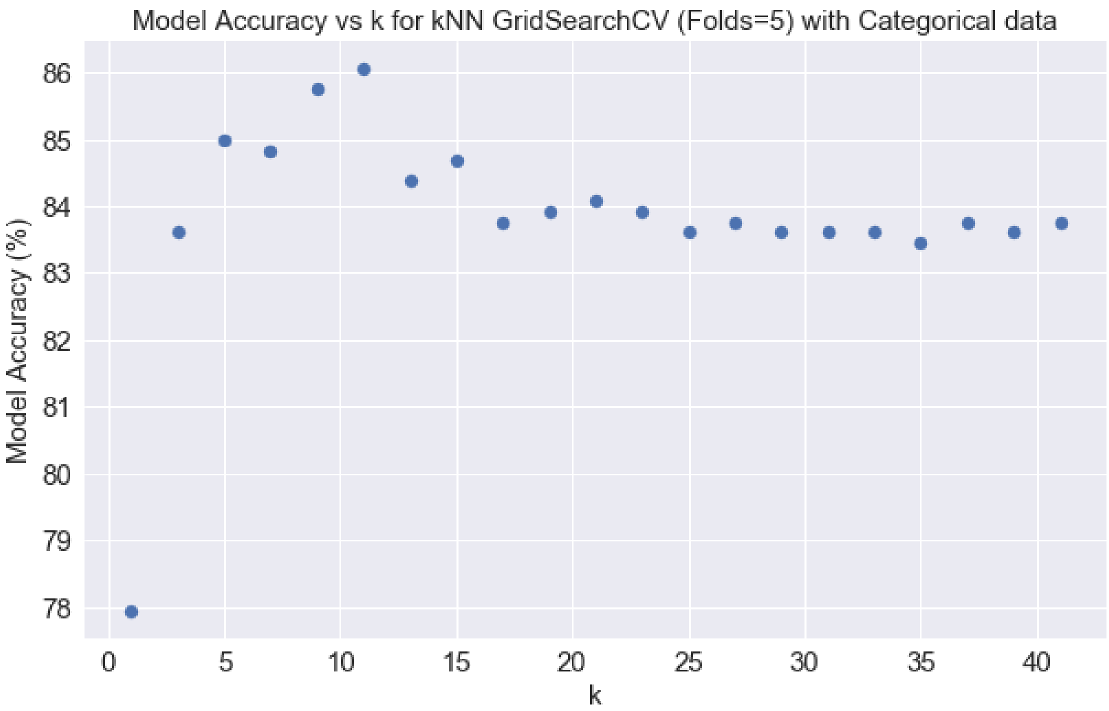
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done 105 out of 105 | elapsed:    0.0s finished
```

[43]:

```
knn_num_cat_results = pd.DataFrame(gs_knn_num_cat_results.cv_results_)
#knn_num_cat_results.head()
```

[44]:

```
plt.figure(figsize=[10, 6])

plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
#plt.ylim(68, 78)

plt.scatter(x=grid_params['n_neighbors'],␣
 ↪y=knn_num_cat_results['mean_test_score']*100)

plt.title('Model Accuracy vs k for kNN GridSearchCV (Folds=5) with Categorical␣
 ↪data', fontsize=15)
plt.ylabel('Model Accuracy (%)', fontsize=15)
plt.xlabel('k', fontsize=15)

plt.show()
```



[45]:
```
print('Best k-NN results: ', gs_knn_num_cat_results.best_score_,␣
 ↪gs_knn_num_cat_results.best_params_)
```

```
Best k-NN results:  0.8606106870229008 {'n_neighbors': 11}
```

Great, adding in the categorical data has improved the k-NN model accuracy to around 86%! It has also reduced value of k required to get the best results, a k of 11 compared to 25 last time gives the best accuracy. Clearly there is a lot of additional predictive power in the categorical features. Lets try the decision tree now.

8.0.2 Decision Tree:

[46]:
```
# Fit the Grid Search CV
gs_tree_num_cat_results = gs_tree.fit(X_num_cat, y)
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```

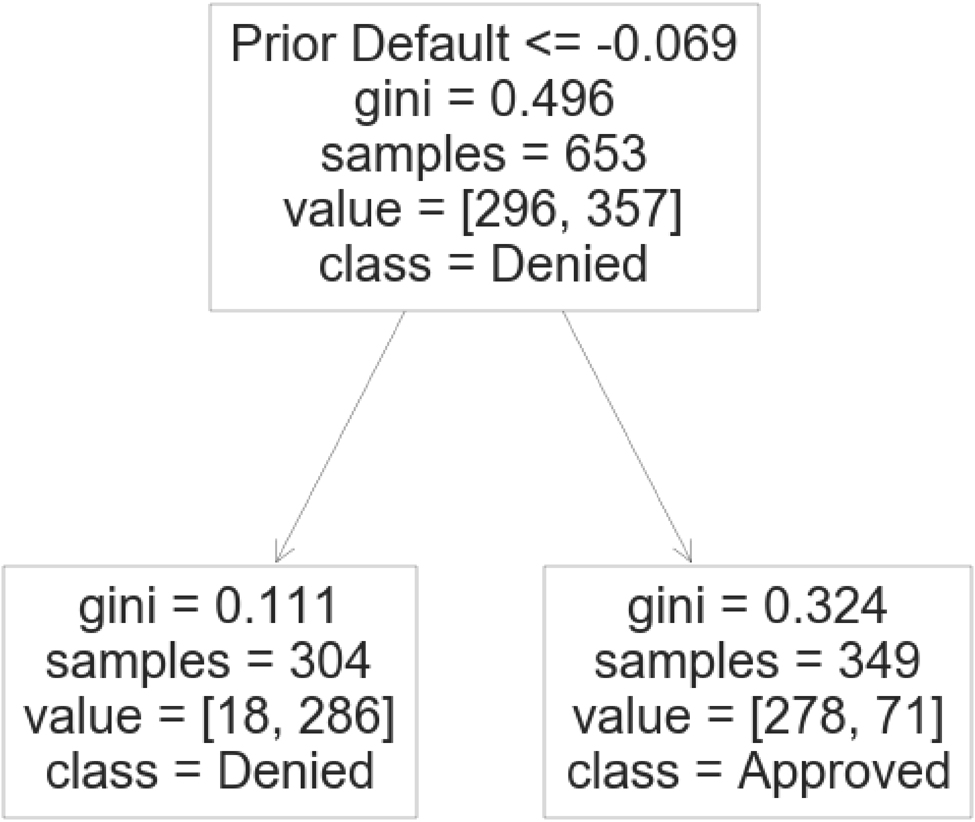
```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done  90 out of  90 | elapsed:    0.0s finished
```

[47]:
```
tree_num_cat_results = pd.DataFrame(gs_tree_num_cat_results.cv_results_)
print(gs_tree_num_cat_results.best_score_, gs_tree_num_cat_results.best_params_)
#tree_num_cat_results.head()
```

```
0.8636758661186142 {'max_depth': 1, 'max_leaf_nodes': 4, 'random_state': 42}
```

Adding in the categorical data has improved the ML model to have an accuracy of around 86.4%! The best model now only has a max depth of 1 - lets look at the tree plot:

```
[48]: plt.figure(figsize=[10, 10])
      plot_tree(gs_tree_num_cat_results.best_estimator_,␣
       ↪feature_names=num_cat_features, class_names=['Approved', 'Denied'])
      plt.show()
```



So this very simple decision tree only uses the 'Prior Default' feature to classify applicants with 86% accuracy! This is a strong indication that many credit applications were possibly decided entirely on this particular feature alone. Clearly if you wanted to apply for a credit card from this particular bank, you were unlikely to be approved if you had a 'Prior Default'. We could train the decision tree only on the 'Prior Default' feature and it would still have 86% accuracy. Finally lets try logistic regression:

8.0.3 Logistic Regression:

```
[49]: # Fit the Grid Search CV
      gs_log_num_cat_results = gs_log.fit(X_num_cat, y)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:    0.0s finished
```

```
[50]: log_num_cat_results = pd.DataFrame(gs_log_num_cat_results.cv_results_)
      #log_num_cat_results.head()
```

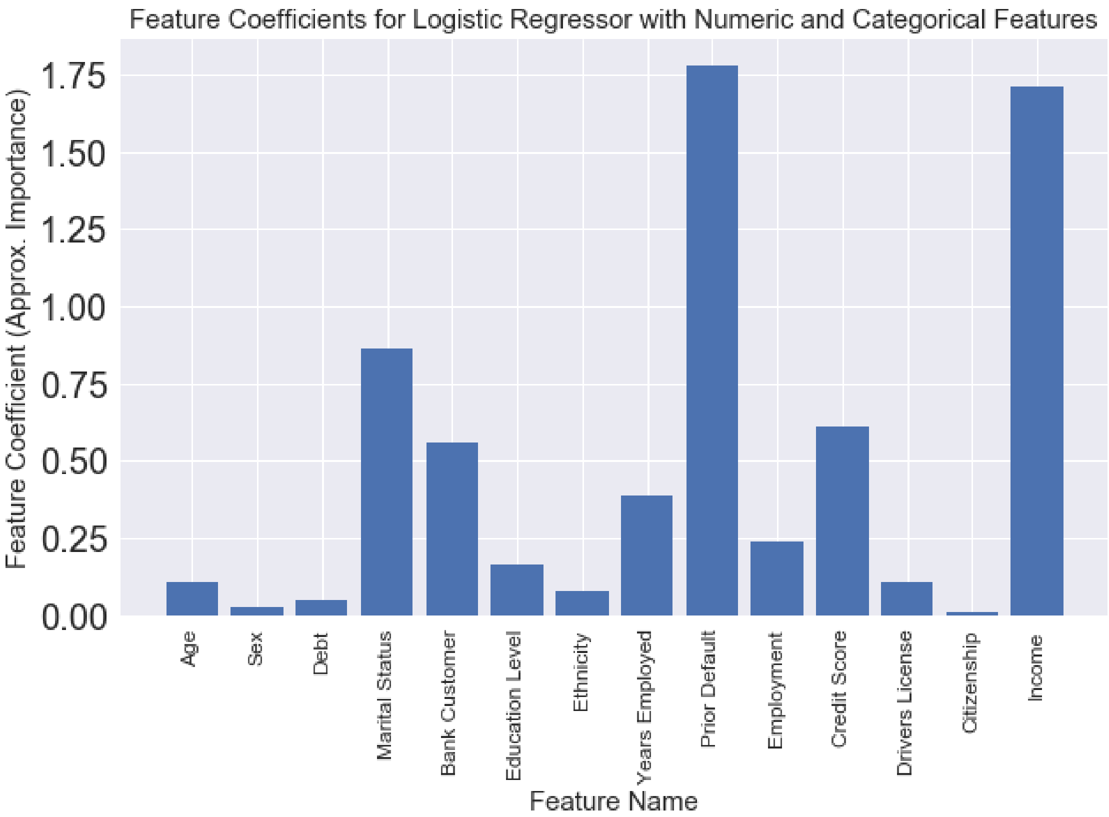
```
[51]: print(gs_log_num_cat_results.best_score_, gs_log_num_cat_results.best_params_)
```

```
0.8575807398708163 {'max_iter': 100, 'tol': 0.01}
```

As we did previously, lets see the most important features for the logistic regressor model by using the coefficients:

```
[52]: # Coefficients of the best logreg model (Negate the features because due to the␣
       ↪approval labels they are negative values):
      num_cat_coefs = [abs(x) for x in gs_log_num_cat_results.best_estimator_.
       ↪coef_[0]]
      print('Regressor Coefs: ', num_cat_coefs)
      plt.figure(figsize=[10, 6])

      plt.yticks(fontsize = 20)
      plt.xticks(fontsize = 12, rotation=90)

      plt.bar(x=num_cat_features, height=num_cat_coefs)

      plt.title('Feature Coefficients for Logistic Regressor with Numeric and␣
       ↪Categorical Features', fontsize=15)
      plt.ylabel('Feature Coefficient (Approx. Importance)', fontsize=15)
      plt.xlabel('Feature Name', fontsize=15)

      plt.show()
```

```
Regressor Coefs:  [0.10921907849759592, 0.030317050778622106,
0.05029762072941195, 0.8623550983231177, 0.5593683248580639,
0.16395622447075944, 0.07844817927720185, 0.38802018112692654,
1.777757443888738, 0.23963586089672212, 0.6105484124154179, 0.10799984643430877,
0.011849131612921171, 1.7086839787319523]
```



Lets quickly look at accuracy, sensitivity, and specificity scores for these new models:

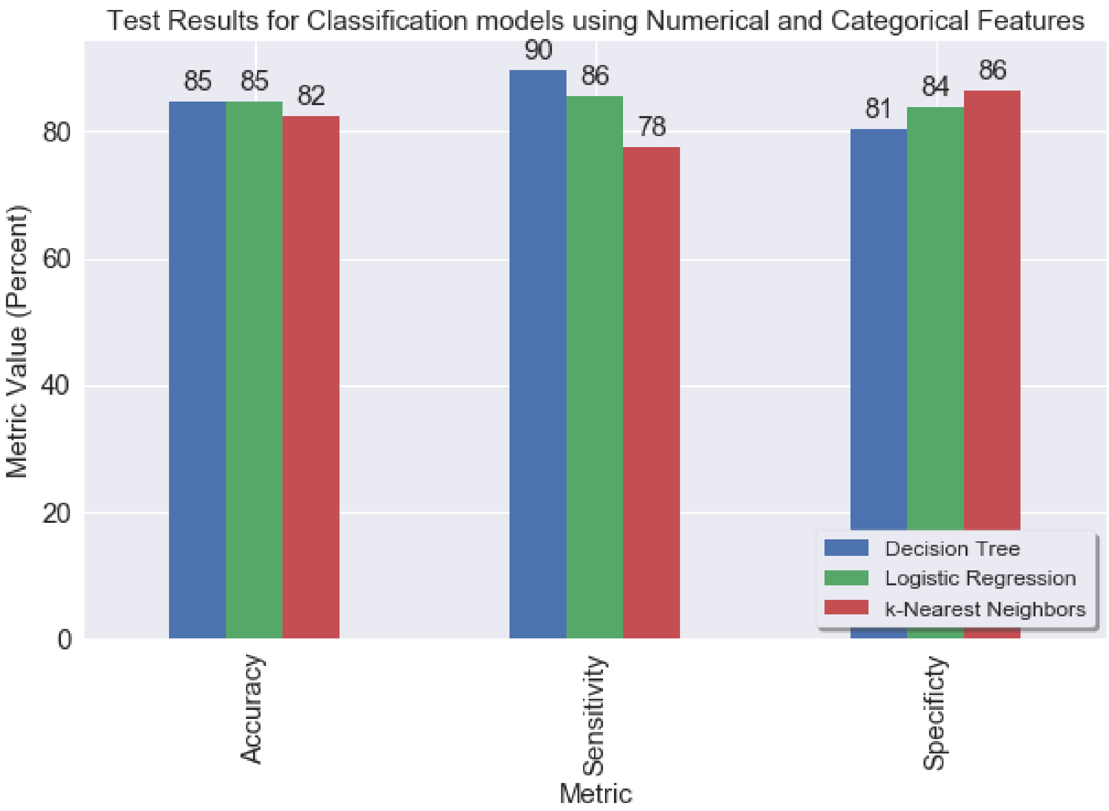
[53]:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix

# Carry out stratified train test split (keep ratio of approved and denied␣
 ↪similar)
X_cat_train, X_cat_test, y_cat_train, y_cat_test = train_test_split(X_num_cat,␣
 ↪y, stratify=y, random_state=42, test_size=0.33)

models_2 = [('k-Nearest Neighbors', KNeighborsClassifier(n_neighbors=5)),␣
 ↪('Decision Tree', DecisionTreeClassifier(max_depth=1, max_leaf_nodes=4,␣
 ↪random_state=42)), ('Logistic Regression', LogisticRegression(max_iter=100,␣
 ↪tol=0.01))]
results_2 = list()

# Iterate through the models:
for name, model in models_2:

    # Get model accuracy
    model.fit(X_cat_train, y_cat_train)
    y_pred = model.predict(X_cat_test)
    acc = model.score(X_cat_test, y_cat_test)
    print(f'{name} accuracy: ', acc*100)
    results_2.append(('Accuracy', name, acc*100))

    # Get confusion matrix
    print(f'{name} confusion matrix:')
    c_m = confusion_matrix(y_num_test, y_pred)
    print(c_m)

    # Get sensitivity and specificity
    tpr = c_m[0][0]/(c_m[0][0] + c_m[0][1])
    tnr = c_m[1][1]/(c_m[1][1] + c_m[1][0])
    print('True Positive Rate: ', tpr*100)
    print('True Negative Rate: ', tnr*100)
    results_2.append(('Sensitivity', name, tpr*100))
    results_2.append(('Specificty', name, tnr*100))
    #plot_confusion_matrix(model, X_num_test, y_num_test)

    print('')

print(results)
```

```
k-Nearest Neighbors accuracy:  82.4074074074074
k-Nearest Neighbors confusion matrix:
[[ 76  22]
 [ 16 102]]
True Positive Rate:  77.55102040816327
True Negative Rate:  86.4406779661017

Decision Tree accuracy:  84.72222222222221
Decision Tree confusion matrix:
[[88 10]
 [23 95]]
True Positive Rate:  89.79591836734694
True Negative Rate:  80.50847457627118

Logistic Regression accuracy:  84.72222222222221
Logistic Regression confusion matrix:
[[84 14]
 [19 99]]
True Positive Rate:  85.71428571428571
True Negative Rate:  83.89830508474576

[('Accuracy', 'k-Nearest Neighbors', 75.92592592592592), ('Sensitivity',
'k-Nearest Neighbors', 52.04081632653062), ('Specificty', 'k-Nearest Neighbors',
95.76271186440678), ('Accuracy', 'Decision Tree', 77.77777777777779),
('Sensitivity', 'Decision Tree', 58.16326530612245), ('Specificty', 'Decision
Tree', 94.0677966101695), ('Accuracy', 'Logistic Regression',
76.85185185185185), ('Sensitivity', 'Logistic Regression', 60.204081632653065),
('Specificty', 'Logistic Regression', 90.67796610169492)]
```

[54]:

```
num_comp_data = pd.DataFrame(results_2, columns=['Metric', 'Model', 'Result'])

num_comp_data = num_comp_data.set_index(['Metric', 'Model'])

plot = num_comp_data.unstack().plot(kind='bar', figsize=[10,6], y='Result')

for p in plot.patches:
    plot.annotate(str(int(p.get_height().round(0))), xy=(p.get_x() + p.
 ↪get_width() / 2, p.get_height()),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=15)

plot.set_title(f'Test Results for Classification models using Numerical and␣
 ↪Categorical Features', fontsize=15)
plot.set_ylabel('Metric Value (Percent)', fontsize=15)
plot.set_xlabel(f'Metric', fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plot.legend(frameon=True, framealpha=1, shadow=True, fontsize=12, loc='lower␣
 ↪right')

plt.show()
```



8.0.4 Simplifying Features:

We can see from the plot above that for the logistic regressor, Marital Status, Bank Customer, Years Employed, Prior Default, Credit Score and Income are the most important features, with Prior Default again being the most important, with Income a close second. How much worse would the logistic model be if we limited the features to those with coefficients above 0.25 in the above plot?

[55]:
```
selected_features = ['Marital Status', 'Bank Customer', 'Years Employed',␣
↪'Employment', 'Prior Default', 'Credit Score', 'Income']

X_simple = scaled_data[selected_features].values


gs_log_simple_results = gs_log.fit(X_simple, y)
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:    0.0s finished
```

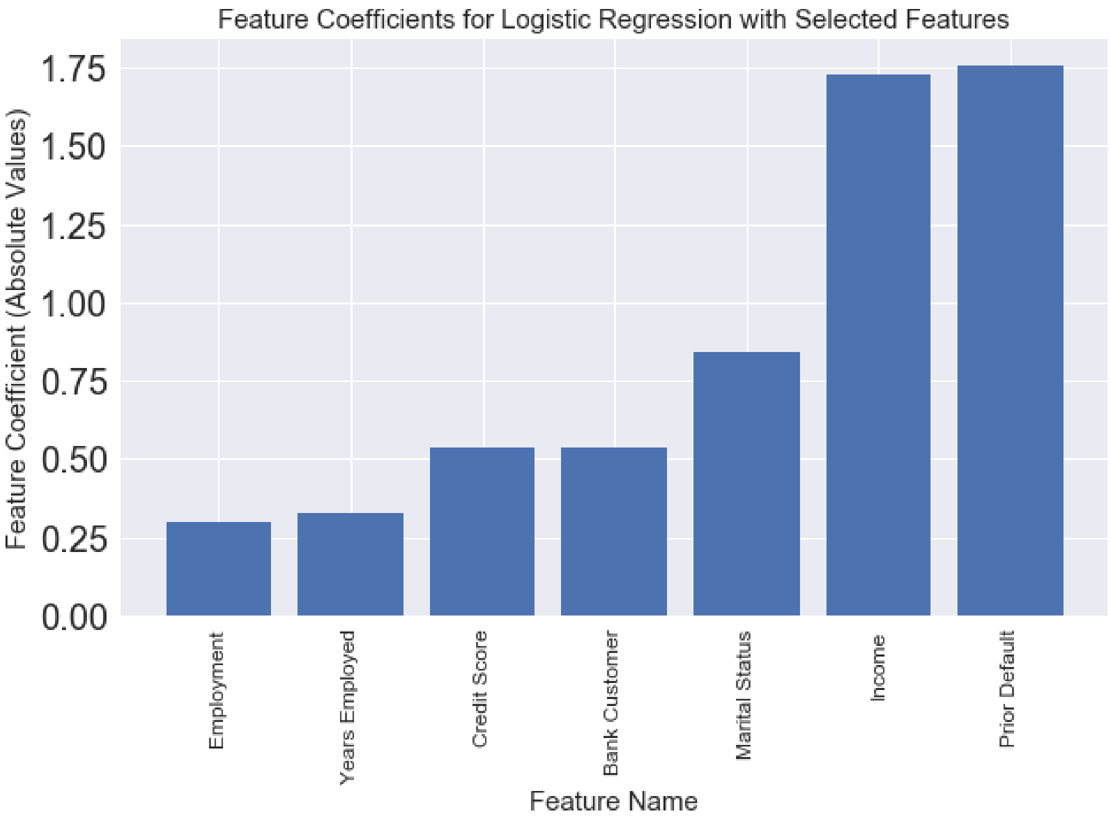
[56]:
```
print(gs_log_simple_results.best_score_, gs_log_simple_results.best_params_)
```

```
0.8637110980622431 {'max_iter': 100, 'tol': 0.01}
```

[57]:

```
# Coefficients of the best logreg model (take absolute value of the␣
↪coefficients):
simple_coefs = [abs(x) for x in gs_log_simple_results.best_estimator_.coef_[0]]
simple_coefs_df = pd.DataFrame(tuple(zip(selected_features, simple_coefs)),␣
↪columns=['Feature Name', 'Value']).sort_values('Value')
print(simple_coefs_df)
print('Regressor Coefs: ', simple_coefs)
plt.figure(figsize=[10, 6])

plt.yticks(fontsize = 20)
plt.xticks(fontsize = 12, rotation=90)

plt.bar(x=simple_coefs_df['Feature Name'], height=simple_coefs_df['Value'])

plt.title('Feature Coefficients for Logistic Regression with Selected␣
↪Features', fontsize=15)
plt.ylabel('Feature Coefficient (Absolute Values)', fontsize=15)
plt.xlabel('Feature Name', fontsize=15)

plt.show()
```

```
     Feature Name     Value
3      Employment  0.298792
2  Years Employed  0.325810
5    Credit Score  0.535364
1   Bank Customer  0.537524
0  Marital Status  0.840017
6          Income  1.730030
4   Prior Default  1.755284
Regressor Coefs:  [0.8400172378092982, 0.5375236057655203, 0.325809684166778,
0.29879190278976253, 1.7552843112157137, 0.5353641327199113, 1.7300302987603753]
```



This has slightly improved the model accuracy to around 86%, very similar to that of the Decision Tree. Note that if you reduce the features down to only 'Prior Default' this very slightly reduces the model accuracy (identical accuracy to the Decision Tree as might be expected!). Can we also reduce the complexity of the k-NN model using while maintaining accuracy using these features?

[58]:
```
# Fit the kNN Grid Search CV
gs_knn_simple_results = gs.fit(X_simple, y)
```

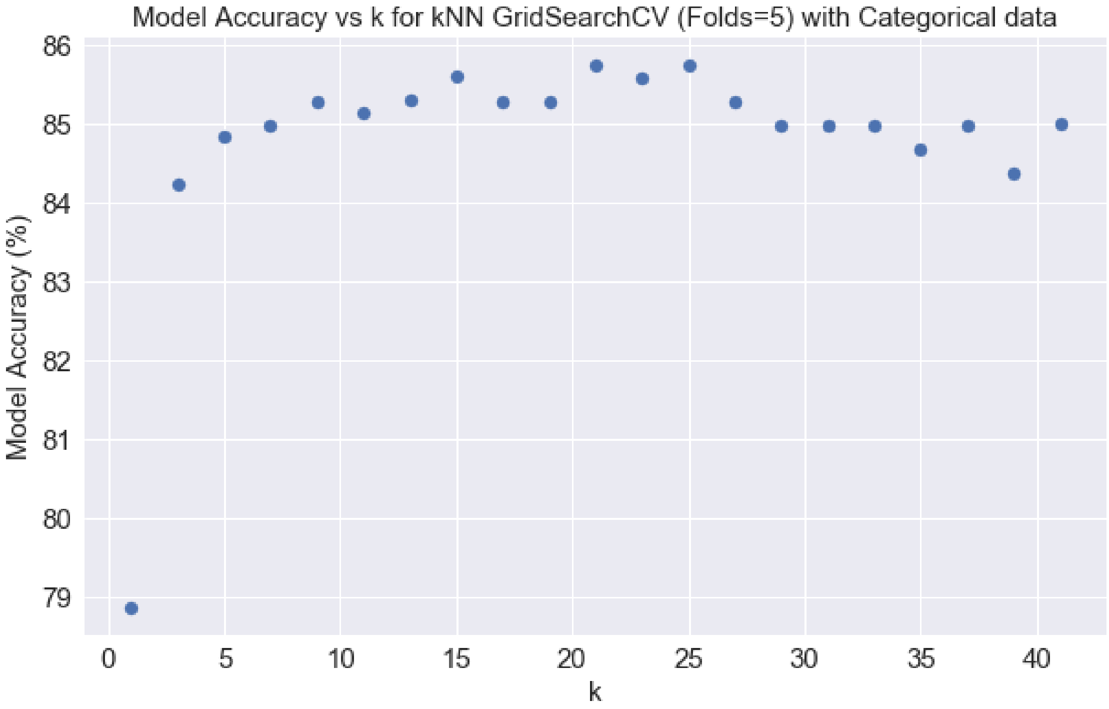
```
Fitting 5 folds for each of 21 candidates, totalling 105 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:    0.0s
[Parallel(n_jobs=-1)]: Done 105 out of 105 | elapsed:    0.0s finished
```

[59]:
```
knn_simple_results = pd.DataFrame(gs_knn_simple_results.cv_results_)
print(gs_knn_simple_results.best_score_, gs_knn_simple_results.best_params_)
#knn_simple_results.head()
```

```
0.8575337639459777 {'n_neighbors': 21}
```

[60]:
```
plt.figure(figsize=[10, 6])
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
#plt.ylim(68, 78)

plt.scatter(x=grid_params['n_neighbors'],␣
 ↪y=knn_simple_results['mean_test_score']*100)

plt.title('Model Accuracy vs k for kNN GridSearchCV (Folds=5) with Categorical␣
 ↪data', fontsize=15)
plt.ylabel('Model Accuracy (%)', fontsize=15)
plt.xlabel('k', fontsize=15)

plt.show()
```



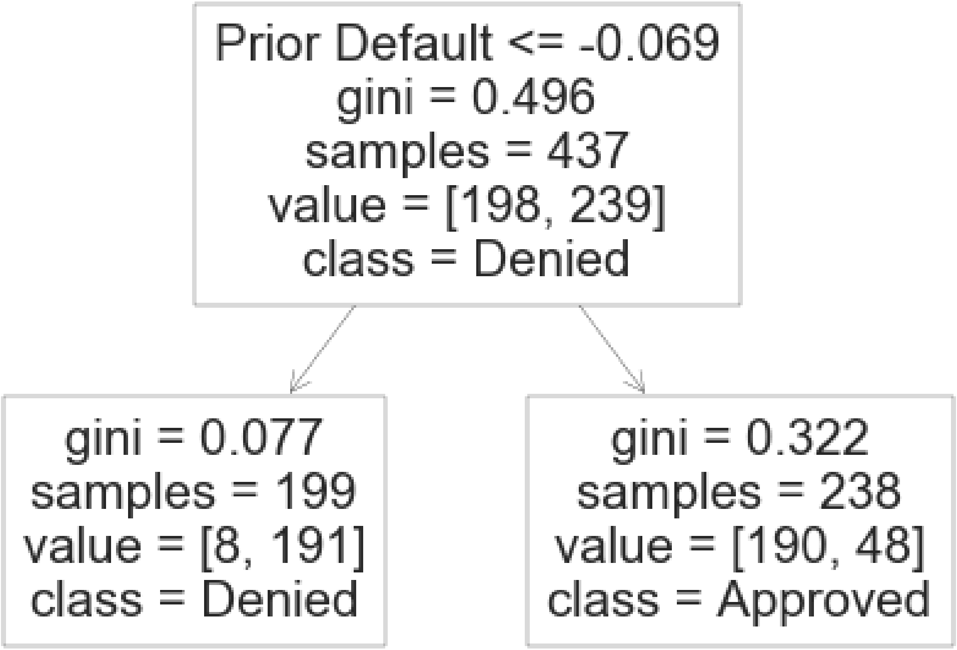
Compared to the mean accuracy using all the features (86%) the accuracy of the model using the set of more important features only only has a very slightly reduced accuracy (85% at k = 5 and 86% at k = 21), showing again that the features removed were not key indicators when classifying the credit applications. Lets retrain the best models from each classifier using only the selected features, to determine accuracy, sensitivity and specificity again:

[61]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
```

```python
# Carry out stratified train test split (keep ratio of approved and denied␣
↪similar)
X_simple_train, X_simple_test, y_simple_train, y_simple_test =␣
↪train_test_split(X_simple, y, stratify=y, random_state=42, test_size=0.33)

models_3 = [('k-Nearest Neighbors', KNeighborsClassifier(n_neighbors=21)),␣
↪('Decision Tree', DecisionTreeClassifier(max_depth=1, max_leaf_nodes=4,␣
↪random_state=42)), ('Logistic Regression', LogisticRegression(max_iter=100,␣
↪tol=0.01))]
results_3 = list()

# Iterate through the models:
for name, model in models_3:

    # Get model accuracy
    model.fit(X_simple_train, y_simple_train)
    y_pred = model.predict(X_simple_test)
    acc = model.score(X_simple_test, y_simple_test)
    print(f'{name} accuracy: ', acc*100)
    results_3.append(('Accuracy', name, acc*100))

    # Get confusion matrix
    print(f'{name} confusion matrix:')
    c_m = confusion_matrix(y_simple_test, y_pred)
    print(c_m)

    # Get sensitivity and specificity
    tpr = c_m[0][0]/(c_m[0][0] + c_m[0][1])
    tnr = c_m[1][1]/(c_m[1][1] + c_m[1][0])
    print('True Positive Rate: ', tpr*100)
    print('True Negative Rate: ', tnr*100)
    results_3.append(('Sensitivity', name, tpr*100))
    results_3.append(('Specificty', name, tnr*100))
    #plot_confusion_matrix(model, X_num_test, y_num_test)

    # Check that decision tree is still the same when trained on 66% of dataset␣
↪- it is!
    if name=='Decision Tree':
        plot_tree(model, feature_names=selected_features,␣
↪class_names=['Approved', 'Denied'])
        plt.show()

    print('')
```

```
k-Nearest Neighbors accuracy:  84.72222222222221
k-Nearest Neighbors confusion matrix:
[[84 14]
```

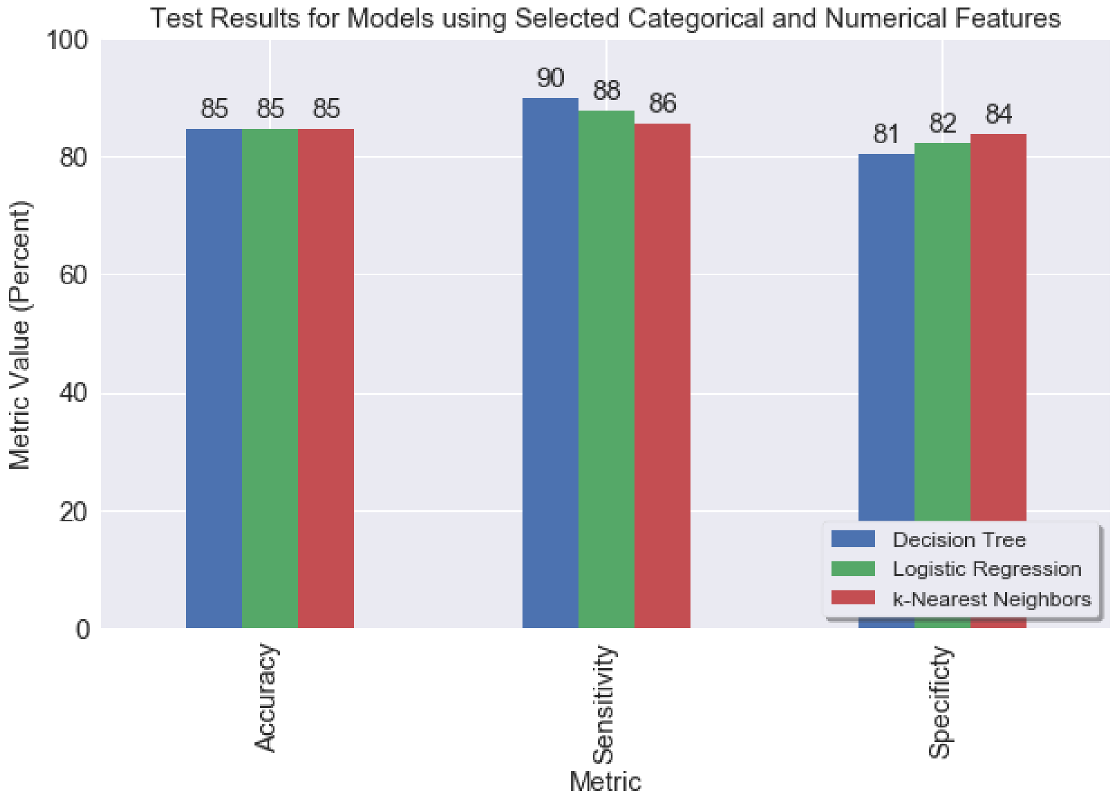
```
 [19 99]]
True Positive Rate:  85.71428571428571
True Negative Rate:  83.89830508474576

Decision Tree accuracy:  84.72222222222221
Decision Tree confusion matrix:
[[88 10]
 [23 95]]
True Positive Rate:  89.79591836734694
True Negative Rate:  80.50847457627118
```



```
Logistic Regression accuracy:  84.72222222222221
Logistic Regression confusion matrix:
[[86 12]
 [21 97]]
True Positive Rate:  87.75510204081633
True Negative Rate:  82.20338983050848
```

[62]:
```
simple_comp_data = pd.DataFrame(results_3, columns=['Metric', 'Model',␣
↪'Result'])
```

```
simple_comp_data = simple_comp_data.set_index(['Metric', 'Model'])

plot = simple_comp_data.unstack().plot(kind='bar', figsize=[10,6], y='Result')

for p in plot.patches:
    plot.annotate(str(int(p.get_height().round(0))), xy=(p.get_x() + p.
 ↪get_width() / 2, p.get_height()),
                    xytext=(0, 3),  # 3 points vertical offset
                    textcoords="offset points",
                    ha='center', va='bottom', fontsize=15)

plot.set_title(f'Test Results for Models using Selected Categorical and␣
 ↪Numerical Features', fontsize=15)
plot.set_ylabel('Metric Value (Percent)', fontsize=15)
plot.set_xlabel(f'Metric', fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(np.arange(0, 120, 20), fontsize=15)
plot.legend(frameon=True, framealpha=1, shadow=True, fontsize=12, loc='lower␣
 ↪right')

plt.show()
```



We can see from the data above that for the models trained on the selected features have a higher overall accuracy, however this mostly comes from an improved sensitivity (correctly identifying approved applications) and for all models comes with a decrease in the specificity (i.e. all the models now identify more applications that should be denied as approved).

Removing the less significant features from the dataset has helped improve the accuracy score of the kNN classifier in particular. Clearly these extra features with low predictive power were separating samples in the feature space that in reality should have been closer together.

Hence while the overall accuracy of these models is ~10% higher than the models using the numerical features alone, all these models would approve more applications which should otherwise be denied. If the bank would much prefer not to have 'bad credit' customers approved for a credit card, the model using only the numerical features might actually be preferred, however it would result in many more 'good credit' customers being denied a credit card.

9 Conclusions:

The UCL MLR Credit Approval Data Set has been analysed using classification techniques.

Question 1: We visualised the numerical and categorical features of the dataset.:

We first visualised the data: - Of the categorical features, the 'Prior Default' feature looked to be very important, as it largely separated the samples by their classification. Applicants with prior defaults were generally denied, while those without were approved. The 'Employment' feature was also seen to be quite important, with a True 'Employment' variable associated with a high likelihood of approval. - Of the numerical variables it appeared that 'Credit Score' was possibly an important indicator of approval, with low credit score applicants having fewer approvals. 'Years Employed' and 'Income' also looked like reasonable measures of approval likelihood, with applicants with lower values for these features being approved less often.

We then fit models to the data, and looked at the features that were important for the Decision Tree and Logistic Regression classifiers (see below).

Question 2: We built ML classification models on using first the numerical and then numerical and categorical features of the dataset.:

- kNN, Decision Tree and Logisitic Regression Classifiers were fit to the data:
  - All models were found to have similar accuracy during GridSearchCV testing, with around 75%-76% accuracy for the numeric features only, and 85-86% accuracy when using the numeric and categorical features.
  - When using only the numeric features, Income, Credit Score, Years Employed and Debt were found to be the most important features. The 'Age' feature was found to have no predictive power for both the Decision Tree and Logistic Regression Models.
  - A very simple decision tree model with only one node and two leaves was found to have 86% accuracy. If the applicant had a prior default they were classified as 'Denied' while if the applicant did not have a prior default they were classified as 'approved'.
  - These accuracies are better than random chance, or classifying all samples as 'denied' (which for this dataset would yield an accuracy of ~55% overall), showing that the models do have a predictive capacity.
  - Models trained using selected categorical and numerical features were found to have higher accuracy overall (82-85%) than those using the numerical features alone. However

this improved accuracy came through increased sensitivy (90-78%), and came at the cost of slightly reduced specificity (81-84%). Depending on the Bank's specific requirements, they may prefer to avoid giving credit to 'bad' applications as much as possible, and so may prefer the models using only numeric features, which had higher specificity.

- We retrained and tested the classifiers, using the 7 features deemed most important for classification. These models maintained (and in the case of k-NN improved) their testing accuracy (all 85% accuracy) compared to models trained on all numerical and categorical features, while removing the features with poorer predictive power. This is an indication that these dropped features were not really utilized when making decisions regarding credit approval/denial (and were reducing prediction accuracy for the k-NN model).

Overall it seems if you were applying for credit to this particular bank you would really not want to have a 'Prior Default' on your record!

9.0.1 Limitations:

Some limitations of this work would include:

- The exact results depend greatly on the dataset used. A different bank might have very different criteria for a credit approval, and so the conclusions from the work regarding the importance of e.g. the prior default feature would no longer be valid.

- Some results from this project (e.g. the resulting tree from a Decision Tree classifier) have some aspects of random chance to them and as such the best performing tree will sometimes vary when the model is trained. Where several trees give similar accuracy results, it is probably best to consider the simplest one.

- Similarly the exact results of this analysis depend somewhat on the ordering of the samples in the dataset, if the samples are shuffled, the results are slightly different (i.e. different values of k for kNN classifier produce the best results). However the overall accuracies of each of the models do not vary greatly.

9.0.2 Further Work:

Some ideas for further work would include:

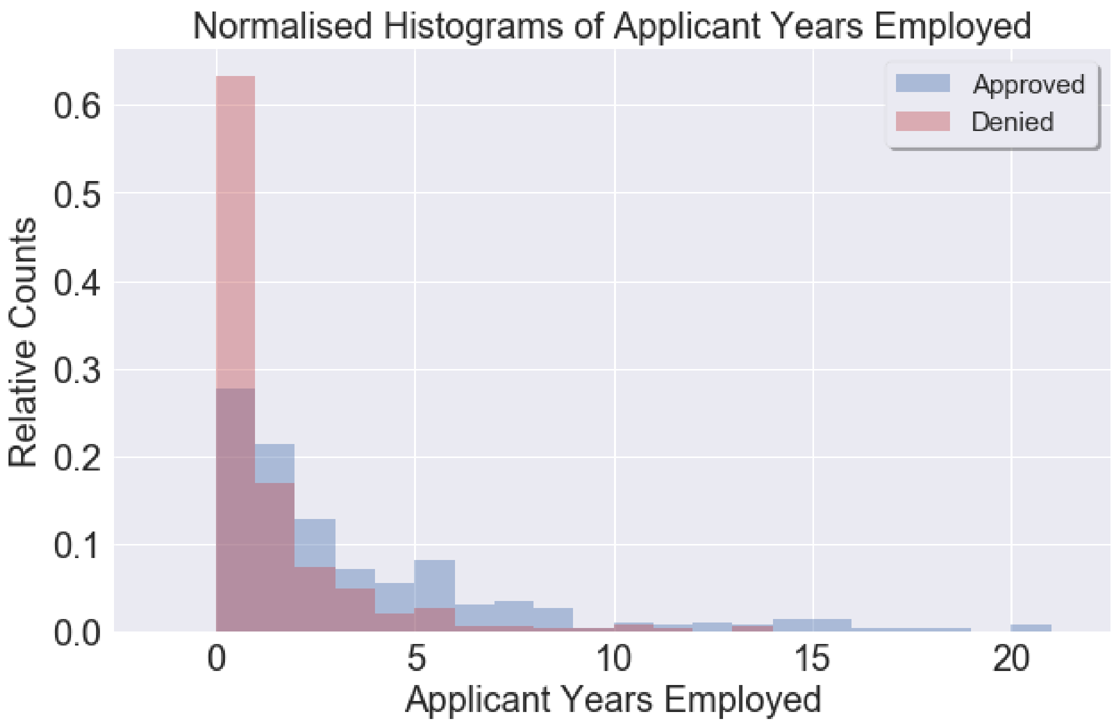
- Performing a similar analysis on other credit approval datasets, such that perhaps differences in approval requirements can be found between providers.

- Using additional classification models, for example neural networks.

[63]:

```
### Extra graphs for presentation:

plt.figure(figsize=[10, 6])

plt.xticks(np.arange(0, 30, 5), fontsize=20)
plt.yticks(fontsize=20)
plt.xlim(-2.55, 22.5)
plt.hist(cleaned_data[cleaned_data['Approval'] == '+']['Years Employed'],␣
 ↪bins=np.arange(0, 31, 1), alpha=0.4, density=True, color='#4c72b0',␣
 ↪label='Approved')
```

```
plt.hist(cleaned_data[cleaned_data['Approval'] == '-']['Years Employed'],␣
 ↪bins=np.arange(0, 31, 1), alpha=0.4, density=True,  color='#c44e52',␣
 ↪label='Denied')
plt.xlabel('Applicant Years Employed', fontsize=20)
plt.ylabel('Relative Counts', fontsize=20)
plt.title('Normalised Histograms of Applicant Years Employed', fontsize=20)
plt.legend(fontsize=15, frameon=True, framealpha=1, shadow=True)
plt.show()
```



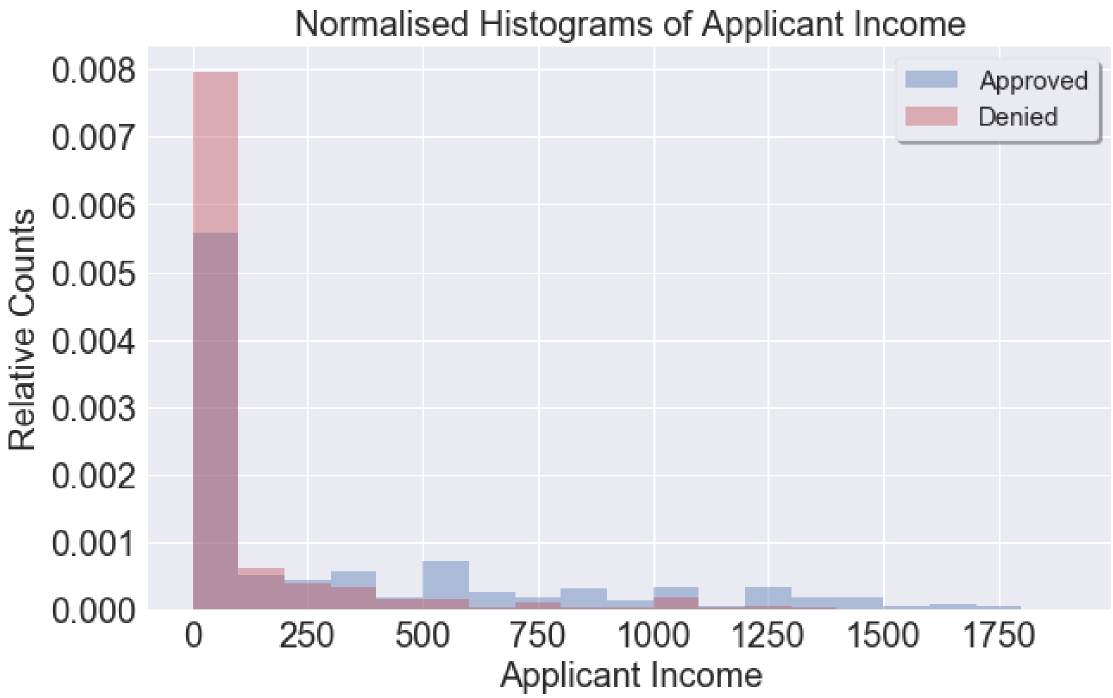
```
[64]: # Use Boolean indexing to get required subsets of data:

zero_score = cleaned_data['Years Employed'] < 1
denied = cleaned_data['Approval'] == '-'
approved = cleaned_data['Approval'] == '+'
print('<1 Years Employed Denied % (Percent of Denials): ', 100 * cleaned_data.
 ↪loc[zero_score & denied].count().max() / cleaned_data.loc[denied].count().
 ↪max())
print('<1 Years Employed Denied % (Percent of Applications): ', 100 *␣
 ↪cleaned_data.loc[zero_score & denied].count().max() / 657)
print('<1 Years Employed Approved % (Percent of Approvals): ', 100*␣
 ↪cleaned_data.loc[zero_score & approved].count().max() / cleaned_data.
 ↪loc[approved].count().max())
```

```
print('<1 Years Employed Approved % (Percent of Applications): ', 100*␣
 ↪cleaned_data.loc[zero_score & approved].count().max() / 657)
```

```
<1 Years Employed Denied % (Percent of Denials):  63.30532212885154
<1 Years Employed Denied % (Percent of Applications):  34.398782343987826
<1 Years Employed Approved % (Percent of Approvals):  27.7027027027027
<1 Years Employed Approved % (Percent of Applications):  12.480974124809741
```

[65]:

```
plt.figure(figsize=[10, 6])

plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
#plt.xlim(-2.55, 22.5)
plt.hist(cleaned_data[cleaned_data['Approval'] == '+']['Income'], bins=np.
 ↪arange(0, 2000, 100), alpha=0.4, density=True, color='#4c72b0',␣
 ↪label='Approved')
plt.hist(cleaned_data[cleaned_data['Approval'] == '-']['Income'], bins=np.
 ↪arange(0, 2000, 100), alpha=0.4, density=True,  color='#c44e52',␣
 ↪label='Denied')
plt.xlabel('Applicant Income', fontsize=20)
plt.ylabel('Relative Counts', fontsize=20)
plt.title('Normalised Histograms of Applicant Income', fontsize=20)
plt.legend(fontsize=15, frameon=True, framealpha=1, shadow=True)
plt.show()
```



[66]:

```
# Use Boolean indexing to get required subsets of data:

low_income = cleaned_data['Income'] < 100
denied = cleaned_data['Approval'] == '-'
approved = cleaned_data['Approval'] == '+'
print('Low Income Denied % (Percent of Denials): ', 100 * cleaned_data.
 ↪loc[low_income & denied].count().max() / cleaned_data.loc[denied].count().
 ↪max())
print('Low Income Denied % (Percent of Applications): ', 100 * cleaned_data.
 ↪loc[low_income & denied].count().max() / 657)
print('Low Income Approved % (Percent of Approvals): ', 100* cleaned_data.
 ↪loc[low_income & approved].count().max() / cleaned_data.loc[approved].
 ↪count().max())
print('Low Income Approved % (Percent of Applications): ', 100* cleaned_data.
 ↪loc[low_income & approved].count().max() / 657)
```

```
Low Income Denied % (Percent of Denials):  77.3109243697479
Low Income Denied % (Percent of Applications):  42.009132420091326
Low Income Approved % (Percent of Approvals):  44.932432432432435
Low Income Approved % (Percent of Applications):  20.243531202435314
```

[67]:

```
# Making Decision Tree values more readable:

# Numerical decision tree values:
# Credit Score 0
# Years Employed -0.289
# Years Employed -0.635
# Income 0.888
# Income -0.102
# Debt 0.407
# Debt -0.692
test = encoded.iloc[0].values
print(scaler.inverse_transform(test[0:15]))
test[10] = 0
print('Credit Score 0: ', scaler.inverse_transform(test[0:15])[10])
test[7] = -0.289
print('Years Employed -0.289: ', scaler.inverse_transform(test[0:15])[7])
test[7] = -0.635
print('Years Employed -0.635: ', scaler.inverse_transform(test[0:15])[7])
test[14] = 0.888
print('Income 0.888: ', scaler.inverse_transform(test[0:15])[14])
test[14] = -0.102
print('Income 0.102: ', scaler.inverse_transform(test[0:15])[14])
test[2] = 0.407
print('Debt 0.407: ', scaler.inverse_transform(test[0:15])[2])
test[2] = -0.692
print('Debt -0.692: ', scaler.inverse_transform(test[0:15])[2])
```

```
#print(test)
#print(encoded.iloc[0])
#print(scaled_data.iloc[0])
#print(scaler.inverse_transform(scaled_data.iloc[0, 0:15]))
```

```
[1.15197799e+00 3.96198022e+02 4.82953292e+00 1.65757393e+00
 4.68606432e-01 5.72888507e+01 2.25307983e+01 6.45496808e+00
 1.03326770e+00 9.35837421e-01 7.46698812e+00 4.62480858e-01
 1.65390505e-01 3.41502751e+04 1.01376110e+03]
Credit Score 0:  2.5022970903522204
Years Employed -0.289:  1.2707880727644612
Years Employed -0.635:  0.105273919743O4212
Income 0.888:  5675.09913779465
Income 0.102:  478.33713910166796
Debt 0.407:  6.8739862105584395
Debt -0.692:  1.353460016772691
```

[ ]:

```

```