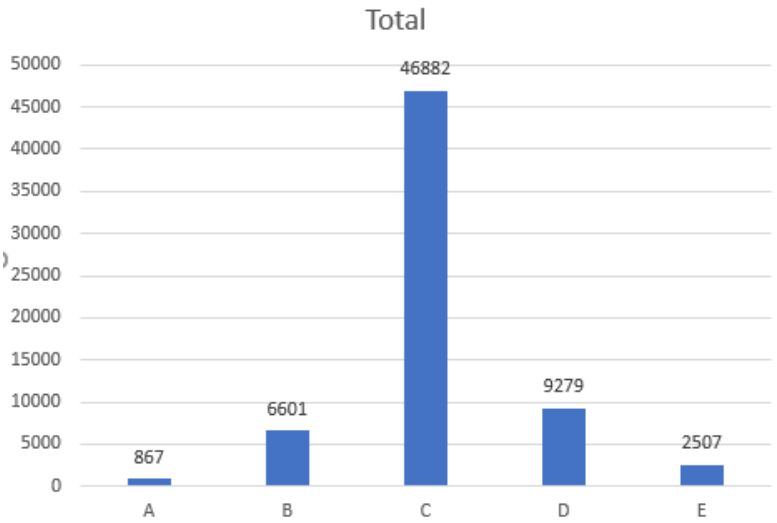
Initial Data Analysis

In order to have a nice overview of the whole data in rows and columns, I imported the data to Excel. With a whopping 296 columns, feature selection is necessary to reduce the dimensions. In addition, oversampling the data points is also necessary because it has imbalanced classes (see chart below).



Import dataset and relevant Python libraries

For the NN model I used the `neural_network` library of Scikit Learn with the estimator Multi-Layer Perceptron Classifier. After importing the csv file as a dataframe, I separated the target variable from the data.

Feature selection

Reducing the amount of features in the dataset x to save training time and avoid overfitting. The techniques performed are as follows:

1. Missing value ratio. Result: There are no value missing in the data points.
2. Low variance data. Filter out features with VarianceThreshold from Scikit Learn and set the threshold to be 0.05, i.e. Columns that have variance less than 0.05 will be removed. Result: 48 features are selected.

Create a new dataset

The target variable has first to be encoded, i.e. categorical values (A, B, C, D, E) are transformed to a numerical encoding (0, 1, 2, 3, 4), respectively. The new dataset is established from the selected columns in dataset x and encoded labels y.

So, the final dataset I will use for my model has a shape of (66137, 49), where the 49th column is the labels.

Splitting the dataset

Now that a new cleaner dataset is done, I am ready to split the data I got to training and test dataset. As the name suggest, I'll use training dataset to build my model from and keep the test dataset untouched and use it at the end to test the model I built. This is quickly done using `train_test_split` from Scikit Learn (test dataset set to be 20% of the whole data).

Last steps of data preprocessing

Back to the second problem I have: imbalanced classes. I opted for the oversampling technique SMOTE to oversample all the classes except the most represented one (class C), so that I have a dataset with

equally-distributed classes (37484 data points per class in training dataset and 9398 data points per class in test dataset).

Then, I scaled the training data to feed to the Neural Network model.

Build the Neural Network model for predictions

Here is where the actual model is built. As mentioned before, I used `MLPClassifier` from Scikit Learn. I choose 3 hidden layers with the same number of neurons as there are features with max. 100 iterations. Then, I used the model to predict the test data.

Evaluate NN model and visualization

Once the prediction on my test target variable is made, I decoded the predicted and actual target variable (y_pred, sm_y_test) back to categorical values. For this, changing the values from float to integer is necessary beforehand. Confusion matrix and classification report from the first run are as follows:

```
>>> print(confusion_matrix(sm_y_test, y_pred))
[[2269 3556  479 1609 1485]
 [ 831 3899 1243 2000 1425]
 [ 103  669 7382  905  339]
 [ 528 1890 1487 3590 1903]
 [ 678 2062  627 3054 2977]]

>>> print(classification_report(sm_y_test,y_pred))
              precision    recall  f1-score   support

           A       0.51      0.24      0.33      9398
           B       0.32      0.41      0.36      9398
           C       0.66      0.79      0.72      9398
           D       0.32      0.38      0.35      9398
           E       0.37      0.32      0.34      9398

   micro avg       0.43      0.43      0.43     46990
   macro avg       0.44      0.43      0.42     46990
weighted avg       0.44      0.43      0.42     46990
```

The model is not perfect. Despite the balanced classes, the under-represented class A was only predicted correctly 24% of the time. Meanwhile, class C was predicted correctly 79% of the time. There are different reasons for this. A low learning rate will cause your model to converge very slowly. A high learning rate will quickly decrease the loss in the beginning but might have a hard time finding a good solution. Weight and bias play a big role in how big the output of each neuron has.

For a better visualization I normalized the confusion matrix and created a heatmap.

Cross Validation

To analyse how well my model predicts across the whole training dataset, I cross-validated my data in 3 folds with `cross_val_score`. Here are the scores:

```
>>> print('Cross-validated scores: ', scores)
Cross-validated scores:  [0.63386955 0.68566627 0.69044341]
>>> print('Accuracy: %0.2f (+/- %0.2f)' % (scores.mean(),scores.std()*2))
Accuracy: 0.67 (+/- 0.05)
```

As you can see, the last fold improved the score of the original model — from 0.633 to 0.69 and the model is 67% accurate in the training data.

Model Improvement

To enhance prediction performance, I used Scikit Learn's hyper-parameter optimization tool `GridSearchCV`. Here I adjusted different set of parameters, ran them all, and took the best ones. I took various parameters for hidden layer sizes as well as the number of neurons, learning rate, solver and alpha. Unfortunately, this was not successful for all the parameters I want to test. The model is too complex to do the exhaustive search on the hardware that I have (the same applied to `RandomizedSearchCV`). Therefore, I reduced the parameter space to only two learning rate parameters (0.0001 and 0.01).

```
>>> print('Best parameters found: ', clf.best_params_)
Best parameters found:  {'learning_rate_init': 0.01}
>>> print('Best estimator found: ', clf.best_estimator_)
Best    estimator    found:       MLPClassifier(activation='relu',    alpha=0.0001,
batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(50, 50, 50), learning_rate='constant',
       learning_rate_init=0.01, max_iter=100, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=None, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
>>> print('Best score found: ',clf.best_score_)
Best score found:  0.6275744317575499
```

As you can see, the mean cross-validated score of the best learning parameter from the grid search (0.627) is less than the best score of the default learning rate (0.722). Therefore, the best result is the normal `mlp` prediction without grid search.

Possible next improvement steps

If I acquired the time and the hardware capacity, I would run further `GridSearchCV` to adjust more the parameters. I would start with hidden_layer_sizes:[(50,50,50),(50,100,50),(100,)], alpha:[0.05, 0.1, 0.25] and learning_rate_init: [0.0001, 0.0005, 0.001, 0.01], as well as higher Kfolds such as 10.

Alternative: NN model with Undersampling

I ran the same Neural Network model but fed with undersampled data using `NearMiss` method from the imblearn library. I obtained 708 data points per class in training dataset and 159 in test data.

```
>>> print(classification_report(nm_y_test,y_pred))
              precision    recall  f1-score   support

           A       0.46      0.38      0.42       159
           B       0.22      0.19      0.21       159
           C       0.40      0.36      0.38       159
           D       0.23      0.28      0.25       159
           E       0.31      0.36      0.33       159

   micro avg       0.32      0.32      0.32       795
   macro avg       0.32      0.32      0.32       795
weighted avg       0.32      0.32      0.32       795
```

It predicts correctly in 32% of the time on average, so it leads to a more suboptimal results than the model with oversampling.