The influence of distribution characteristics and data balancing on classification bias in highly unbalanced data sets

Zekiye Erarslan, Manuel Günther, Artemii Redkin, Matteo Zannini

30.09.2023

1 Introduction

In recent years, the field of machine learning has evolved from an emerging science into a widely applied technology, finding applications across various domains such as business, industry, and scientific research. However, as machine learning techniques have gained prominence, a critical challenge has surfaced known as the class imbalance problem. This problem, characterized by a significant disparity in the number of samples between different classes within a dataset, has profound implications for classification performance and decision-making in real-world applications.

It has become increasingly apparent that imbalanced datasets can lead to suboptimal classification results, prompting researchers to explore solutions to mitigate its impact. The class imbalance problem is widespread, affecting a substantial portion of the data mining community. It is essential to understand that class imbalances can manifest in various application domains, including fraud detection, risk management, text classification, and medical diagnosis. In some cases, these imbalances are inherent to the problem, while in others, they arise due to limitations in data collection processes or the need for human intervention in selecting examples for training.

Both data-level and algorithmic-level solutions have been proposed to address class imbalance. These include resampling techniques such as oversampling and undersampling, adjustments to class-specific costs, threshold tuning, and recognition-based learning approaches. Researchers have dedicated considerable effort to developing and refining these methods, aiming to improve the performance and fairness of machine learning models in the face of imbalanced data.

In this report, we illustrate a comparative analysis aiming to highlight the influence of data balancing and other factors in the process of classifying imbalanced datasets. In addition to assessing the performance of established data-level balancing algorithms, our objective includes the exploration of diverse classification scenarios wherein various factors may exert an influence on classification efficacy.

The report is organized into distinct sections, including Methods, Results, and Discussion. Within the Methods section, we comprehensively elucidate our approach, which comprises a four-step pipeline encompassing data generation, balancing, classification, and output analysis. It's noteworthy to emphasize that each step of this methodology is supported by extensive literature research, ensuring a robust foundation for our approach.

In the Results section, we provide the most significant findings derived from the systematic variation of parameters within our defined pipeline. Through rigorous experimentation, we explore the impact of parameter adjustments on the classification outcomes. To facilitate a comprehensive understanding of our findings, we employ data visualization techniques and provide insightful plots that vividly represent the observed results.

In the final section, the Discussion, we delve into a thorough analysis of the results and draw meaningful conclusions from our study. We closely examine what our findings mean in practical terms and how they can be applied in real-world situations. Additionally, we consider areas where our study could be improved or enhanced for future research. By acknowledging both the strengths

and limitations of our work, we lay the groundwork for future studies to build upon our findings and advance the field of classifying imbalanced datasets.

2 Methods

This section will describe the different balancing methods, classifiers and metrics we used to obtain and assess our results. For each balancer and classifier we will give a short theoretical description as to how it works, what the strengths and weaknesses of the method are, and which challenges we encountered in the implementation.

In the following we will assume that we have data we aggregate with the notation $X = (X_j)_{j=1}^n$ and y where for our binary classification problem $X \in \mathbb{R}^{d \times n}$ is the *feature* or *design matrix* containing all n feature vectors of dimension d and $y \in \{0,1\}^n$ is the *target vector* with binary entries. So in our notation y_j is the class label of feature vector $X_j = x_j$, where we set $y_j = 1$ to mean membership to the minority class and $y_j = 0$ to mean membership to the majority class.

By $\mathcal{D} = \{(x_j, y_j) | 1 \leq j \leq n\}$ we denote the entire dataset and by $\mathcal{D}_{\text{train}}, \mathcal{D}_{\text{test}}$ we denote the train and test subsets respectively. As in any particular step either the training or the testing subset of the data is used, and to avoid clutter in the notation we will still use X and y when talking about either subset without additional indication in the notation. When referring to a subset of samples of class 1 we write $\{x_j^1, y_j^1\}$

2.1 Data balancing

As for the data balancing step, here are some important characteristics that determine the usefulness of a balancer:

- The balancer should be generalisable and applicable to different levels of imbalance and subsequent classification methods;
- It should maintain the actual class structure and represent the minority class pattern accurately;
- It should support a correct class identification.

We hereby summarise the main features of the data-level balancing algorithms that we implemented in the pipeline. Typically, these methods are categorised into three main groups: synthetic samplers, resamplers and hybrid samplers.

2.1.1 SMOTE

SMOTE (Synthetic Minority Oversampling Technique) is probably the most known of the balancing methods. SMOTE is an over-sampling technique which expands the training set $\mathcal{D}_{\text{train}}$ by generating synthetic minority class samples.

Given the parameter k, at every iteration the algorithm does the following steps:

(i) Select a random sample (x_j^1, y_j^1) of the minority class

(ii) For the feature vector x_j^1 find the k nearest minority neighbours $x_{l_1}^1 \dots x_{l_k}^1$

(iii) Select one of these neighbours at random, say x_l^1 and sample a uniform $u \sim \mathcal{U}(0,1)$

(iv) Add $(x, 1)$ with $x = x_j^1 + u(x_j^1 - x_l^1)$ to the training set

until the desired number of minority samples is achieved. This process creates new data points along line segments between minority feature vectors, effectively making the decision boundary of the minority class more inclusive. We decided to include SMOTE in our study as it is a reference method, even though important drawbacks have been detected in previous articles, such as presenting computational complexity quadratic in the size of the minority class and distortion of its distribution due to localisation of the selected target points.

2.1.2 Borderline SMOTE

ADASYN is another synthetic sampler

2.2 Metrics

This section gives a brief overview of the metrics commonly used to assess the performance of a classifier in a binary classification situation. Given the true y-values of a test set and classifiers predictions on the corresponding feature vectors most metrics are based on the so called **confusion matrix**.

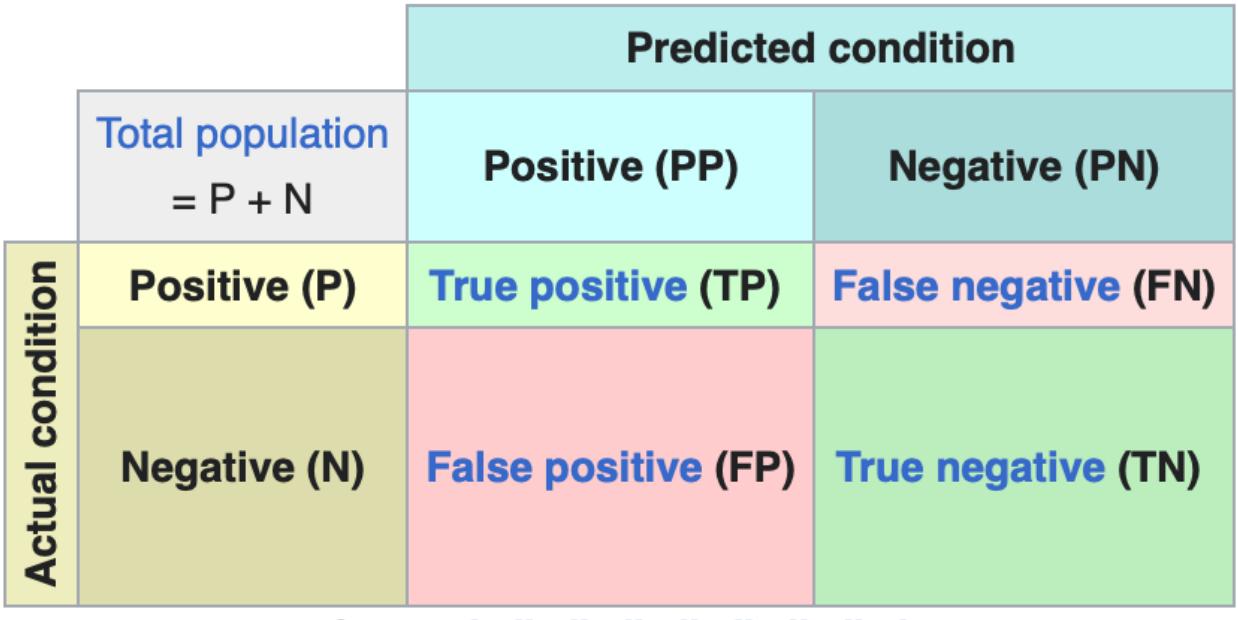

Figure 1: Confusion Matrix

It summarises performance with the number of true positves (TP), true negatives (TN), false positives (FP) and false negatives (FN). What is a positive depends on the target class. In our case this is the class 1 (the minority class) while a negative corresponds to class 0 (the majority class).

From these values many standard measures are derived:

Accuracy: Accuracy is the ratio between all that has been labeled correctly and the total number of samples, i.e.

$$\text{acc} = \frac{TP + TN}{TP + TN + FP + FN}$$

It is a commonly used but fairly basic measure that often fails to accurately represent a classifiers performance in an imbalanced situation, as for a high class imbalance the accuracy is high for a classifier that stubbornly predicts majority class for every sample.

Sensitivity / True Positive Rate / Recall: Recall reflects the fraction of positive samples the classifier has identified i.e.

$$\text{rec} = \frac{TP}{TP + FN}$$

It is more relevant in the imbalanced case as a classifier that chooses to label all before it as majority class to obtain good accuracy will have low recall.

Precision: Precision represents the fraction of samples that have been correctly labeled positive i.e.

$$\text{prec} = \frac{TP}{TP + FP}$$

This measure allows to assess whether the classifier tends to be correct when labelling a sample as minority class.

Specificity: Gives the fraction of samples classified as correctly negative versus all negatives.

$$\text{spec} = \frac{TN}{TN + FP}$$

Specificity is especially important when assessing the frequency and potential cost of false positives.

We have already described four different measures based on the confusion matrix, each of them assessing different aspects of the performance of a classifier. But since all of these contribute important information, and since a classifier can have good scores in one but bad scores in another, how can one give a unified answer to the question "How good is my classifier?" when it comes to evaluation and decision making? This question is the idea for the measures that follow.

One measure that intends to combine at least two of the metrics mentioned above is the **F1-score**. It is the harmonic mean of precision and recall, where the harmonic mean for a set of positive real numbers $x_1, \ldots, x_n$ is given by

$$H(x_1, \cdots x_n) = \frac{n}{\sum_{j=1}^{n} \frac{1}{x_j}}$$

which applied to precision and recall becomes

$$F = \frac{2}{\frac{1}{\text{rec}} + \frac{1}{\text{prec}}} = 2\frac{\text{rec}\,\text{prec}}{\text{rec} + \text{prec}}.$$

There are also weighted versions of the F1-score that are supposed to take into account whether recall or precision are more important in a given situation.

Another important measure that provides a more comprehensive assessment of a classifiers performance is the **Receiver Operator Characteristic (ROC)**. On a theoretical level, supposing here X is the random variable representing the distribution of the feature vectors and our classifier outputs a continuous probability score $s(X)$, the ROC is obtained by considering different cutoff thresholds for that score. Set the threshold τ to decide the predicted class such that if $s(X) > \tau$, we predict the positive class. Then the true positive rate (TPR) and the false positive rate (FPR) are given by

$$TPR(\tau) = \mathbb{P}(s(X) > \tau | Y = 1)$$
$$FPR(\tau) = \mathbb{P}(s(X) > \tau | Y = 0)$$

where Y is the r.v. representing the true class of X for every realisation. TPR is the probability that the classifier ranks a randomly chosen positive instance higher than a randomly chosen negative instance. Similarly, FPR is the probability that the classifier ranks a randomly chosen negative instance higher than a randomly chosen positive instance.

The ROC curve plots $TPR(\tau)$ against $FPR(\tau)$ for all possible thresholds τ, producing a curve that ranges from $(0, 0)$ to $(1, 1)$. We can interpret a ROC plot as plotting the path of a function

$$f : \mathbb{R} \to [0, 1]^2, \quad \tau \mapsto (TPR(\tau), FPR(\tau))$$

The **Area Under the ROC Curve (AUC)** then provides a single scalar value that represents the expected performance of the classifier. An AUC of 1 indicates a perfect classifier, while an AUC of 0.5 indicates a classifier that performs no better than random chance.

AUC can also be interpreted in terms of the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance, assuming that one positive and one negative instance are chosen at random.

In practice ROC curves and AUC values are obtained by sorting the classified samples by score thus AUC is essentially a rank order statistic.

2.3 Net Benefit

The choice of risk threshold implicitly conveys the adopted relative misclassification costs. It can be derived that the odds of the risk threshold equal the harm-to-benefit ratio, which is the harm of a false positive divided by the benefit of a true positive. For example, if a risk threshold of 20% is used, the odds are 1 to 4. Therefore, a 20% risk threshold assumes that the harm of a false positive is one-quarter of the benefit of a true positive or that 1 true positive is worth 4 false positives: A clinician might express this in terms such as "I would not do more than five biopsies to find one cancer." Hence, when applying a model to a set of patients, we can correct the number of true positives (TP) for the number of false positive (FP) using the odds w of the risk threshold t:

$$\text{TP} - w\text{FP} = \text{TP} - \frac{\tau}{1-\tau}\text{FP}$$

When dividing by the total sample size N, the Net Benefit is obtained

$$\text{NB} = \frac{1}{N}(\text{TP} - w\text{FP}) = \frac{1}{N}(\text{TP} - \frac{\tau}{1-\tau}\text{FP})$$

The Net Benefit of treat-none is always 0, whereas the Net Benefit of treat-all is positive for risk thresholds below the event rate and negative for risk thresholds above the event rate.

2.4 Net Benefit and Risk Threshold

Suppose r.v. Y with outcome diseased D or healthy $\neg D$. Let $c_{TP}, c_{FP}, c_{TN}, c_{FN}$ then expected cost of predicting $\neg D$ is

$$\mathbb{P}(Y=1)c_{FN} + \mathbb{P}(Y=0)c_{TN}$$

and expected cost for predicting D is

$$\mathbb{P}(Y=1)c_{TP} + \mathbb{P}(Y=0)c_{FP}$$

If we write $T = \mathbb{P}(Y=1)$ we can rewrite to

$$Tc_{FN} + (1-T)c_{TN}$$

and

$$Tc_{TP} + (1-T)c_{FP}$$

One is indifferent about treatment if both expected costs are equal.

$$\begin{gathered} Tc_{FN} + (1-T)c_{TN} = Tc_{TP} + (1-T)c_{FP} \\ \Leftrightarrow \\ T = \frac{c_{TN} - c_{FP}}{(c_{TN} - c_{FP}) + (c_{TP} - c_{FN})} \end{gathered}$$

It can also be rearranged in a different way

$$\begin{gathered} Tc_{FN} + (1-T)c_{TN} = Tc_{TP} + (1-T)c_{FP} \\ \Leftrightarrow \\ \frac{T}{1-T} = \frac{c_{TN} - c_{FP}}{c_{TP} - c_{FN}} \end{gathered}$$

In the first group, the costs relate to undertreatment (false-negative classifications) The costs of these false-negative classifications c_{FN} should be compared to the costs of true-positive classifications c_{TP}. The difference $c_{TP} - c_{FN}$ is the net benefit of treating all who have the disease compared to treating none of them. Suppose you have a treatment that causes lots of damage as well as curing the disease. Then this difference might be low, i.e. treating everyone with the dangerous treatment does not give much better outcome than simply not treating anyone. Suppose on the other hand you have a devastating disease like Polio and a low cost treatment like a Polio-vaccine, then that difference will be strongly positive.

In the second group, relevant costs are for those without the event if not treated, who are treated ("overtreated"). The costs of these false-positive classifications (c_{FP}) should be compared to the costs of true-negative classifications (c_{TN}) while $c_{TN} - c_{FP}$ is the harm of treating all who don't have the disease compared to the benefit of not treating any of them. E.g. the cost of not treating anyone without the disease might be 0 but the cost of treating them might be high. Then this value is strongly negative. On the other hand suppose again a low impact vaccine that barely does harm, then that difference may be small negative.

Odds (cutoff) = Harm/Benefit

The ratio in case of e.g. the polio vaccine could be strongly in favour of treating everyone (small cost for those without, high benefit for those with disease).

2.5 Calibration

Another key property of a prediction model is calibration, i.e., the agreement between observed outcomes and predictions.

$$TPR(\tau) = \mathbb{P}(s(X) > \tau | Y = 1)$$

2.6 Mixed

$$TPR(\tau) = \mathbb{P}(s(X) > \tau | Y = 1)$$
$$FPR(\tau) = \mathbb{P}(s(X) > \tau | Y = 0)$$

3 Machine Learning Pipeline

This project was focused on two aspects. The first was creating a machine learning pipeline that could facilitate an easy way to conduct experiments on imbalanced data. The second was to use it to conduct some experiments to show it functions and to gain some additional insight into the imbalance problem itself.

Along the way we created three approaches to the pipeline, where the idea with each successive approach was to either improve generality and the potential scope of experiments that could be run, or improve the efficiency of the pipeline computation- or memory-wise to obtain results quicker. The fundamental layout of all three approaches was the same in the sense that each consisted of four or five python classes:

1) A **generator** class that creates the input data for a variety of parameters like the number, dimensionality and imbalance ratio of the samples

2) A **balancer** class that subjects the created data to a chosen balancing method to remedy the class imbalance

3) A **classifier** class that trains a chosen classifier on the output of the balancer class and conducts predictions on the test-set

4) An **metrics** or **assessor** class that applies standard metrics of classification quality to the predictions or guides the flow of the pipeline

The first version of the pipeline is essentially a more elegant packaging for the application of methods from `scikit-learn`, `imblearn`, and some more specialised libraries like `xgboost`. In this version both the balancer and classifier classes are essentially just wrappers that invoke the balancing, training and prediction methods of the imported classes from these libraries. Experiments are conducted directly via for-loop iteration over the experiment parameters.

The second approach was based on two ideas. One is to incorporate a new generator that allows more flexible generation of feature data. The `make_classification` method used in the first approach creates clusters exclusively with Gaussian features and does not allow to specify means or variances. We created an own generator to amplify the control over and the variety of our sample distributions. The other idea was to expand the functionality of the balancer and classifier classes by locating parts of the necessary iterations in a larger experiment in these classes. The hope was to allow brief experiments to be conducted in a more modularised way and improve efficiency. Since the goal of better efficiency was not achieved in this case we created a third approach.

Our third approach was mainly focused on computational efficiency. Stacked for-loop iterations turned out to be fairly slow and required a large amount of time to complete. This is to some extend inevitable as the involved algorithms are complex and the datasets large, but the previous pipelines also created large additional overhead this approach was intended to reduce.

In the following we give a more detailed description of the created pipeline versions.

3.1 First pipeline approach

3.2 Generator Generalisation and Localised Iteration Approach

The Central Limit Theorem and generalisations like Donsker's Theorem provide the theoretical explanation for the reality most practitioners of data analysis are already well aware of: The normal distribution is omnipresent and occurs even in many circumstances one wouldn't expect due to accumulation effects. The already mentioned `make_classification` method recognises this fact and allows to create data from normal distributions of arbitrary dimensions in a simplified way.

There are however also data characteristics that can not be modelled by a normal distribution due to aspects like domain structure or symmetry. `make_classification` also doesn't provide direct control over explanatory distribution parameters like expectation and covariance matrix. We thus created an own generator as well, that was supposed to facilitate greater control over composition and characteristics of feature distributions. The `Multi_Modal_Dist_Generator` was designed, as the name suggests, to create samples with feature-distributions that can have multiple modes. It is designed in such a way that it can sample from any distribution the `scipy` package provides.

In the example below feature vectors with normal and beta distributed features was created and plotted with the `RawVisualiser` class. Here the involved normal distribution is bimodal for the majority and unimodal for the minority class. The first plot shows two normal features 0 and 1 plotted against each other in a scatterplot.

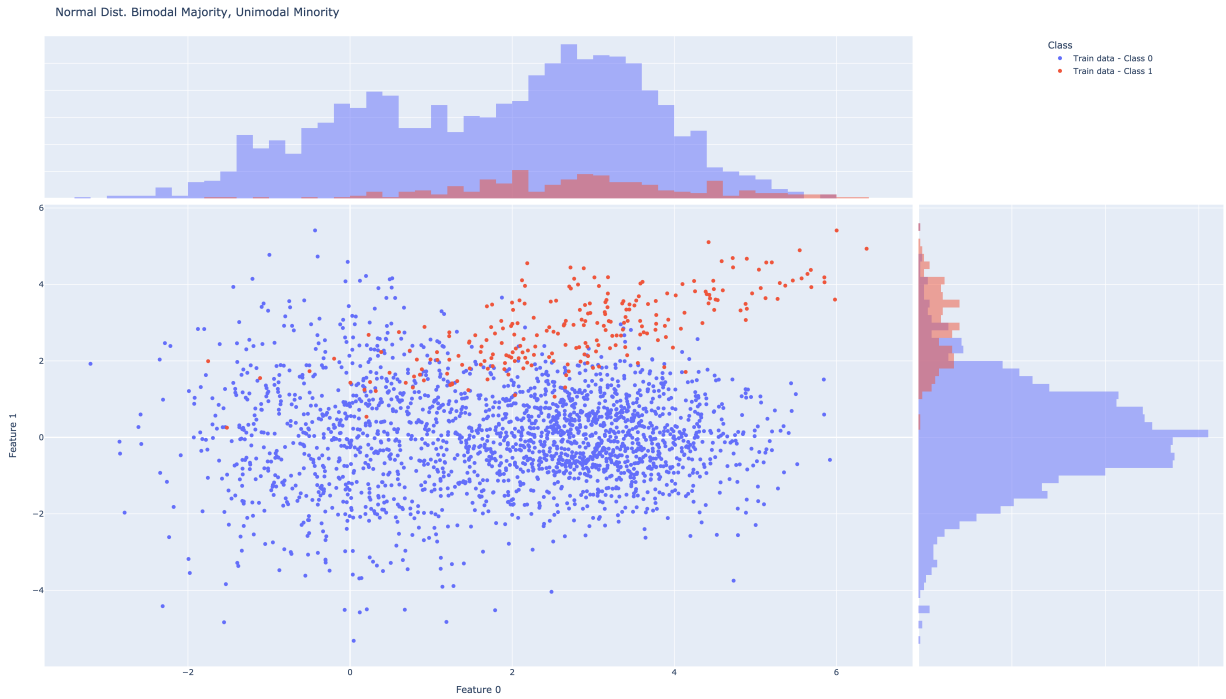

Figure 2: Majority bimodal, minority unimodal-normal features

One can see that different covariance matrices have been used and that a greater variety of distribution shapes and circumstances can be represented with this level of control.

The second figure comes from the same dataset, but plots feature 0 (normal) and feature 2 (beta) against each other.

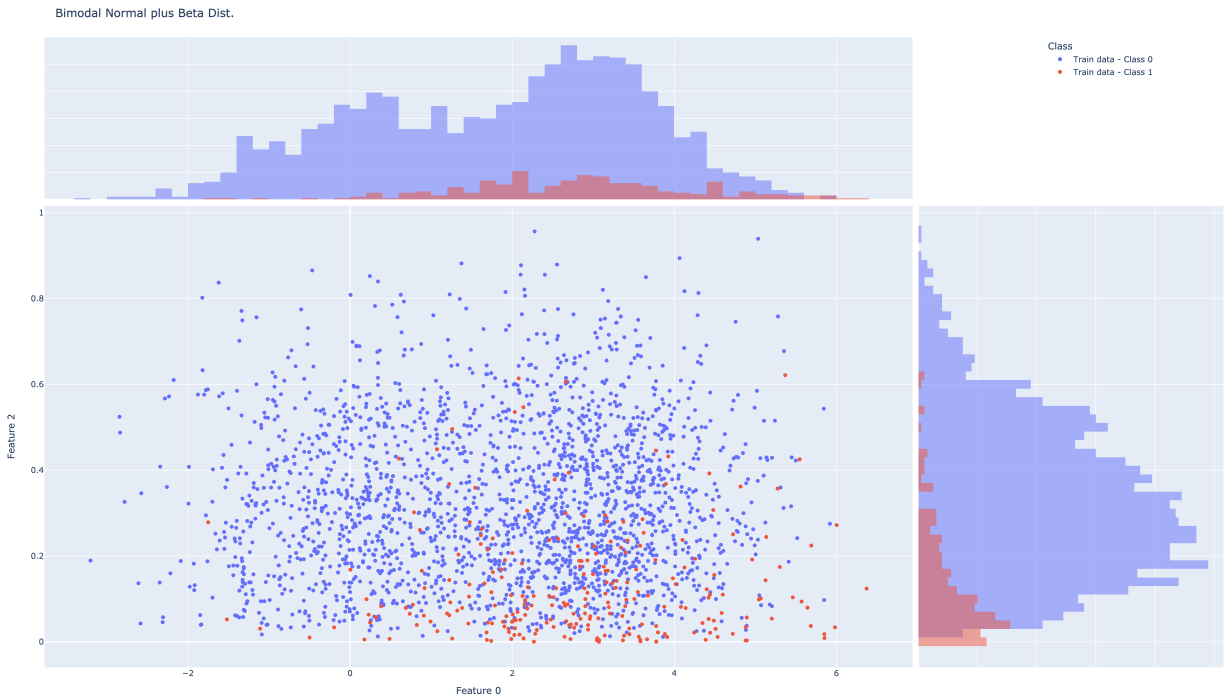

Figure 3: Bimodal-normal (0) and beta (2) features

It is apparent that this generator allows for significantly greater flexibility but at the cost of having to specify all the parameters of every involved distribution. This functionality made it tough

to implement and somewhat tedious to use, as a test that would otherwise only require specifying a number of features or a cluster distance, requires the user to come up with the necessary means, variances, scales or other parameters. On the implementation side this also required to find a way to pass the parameters from a variety of distributions, keep track which parameters belonged to which feature and re-assemble the complete feature vector before doing the train-test-split.

The parameters used to generate the samples represented in the pictures above are given by this code:

```
#set the parameter dictionary for the MV normal. sigma is the standard deviation
mu_c0_1 = [0,0]
mu_c0_2 = [3,0]
mu_c1 = [3,3]
sigma_c0_1 = np.array([[1,0],
                                        [0,3]])
sigma_c0_2 = np.array([[1,0],
                                        [0,1]])
sigma_c1 = np.array([[2,1],
                                    [1,1]])

distributions = [st.multivariate_normal, st.beta]

#set the parameter dictionaries as a list of dictionaries with parameter dictionaries for
    classes individually.
dist_parameter_dicts = [
    {'modes_c0': 2,
          'modes_c1': 1,
          'mixing_weights_c0': [0.4, 0.6],
          'params_c0': {'mean': [mu_c0_1, mu_c0_2], 'cov': [sigma_c0_1, sigma_c0_2]},
          'params_c1': {'mean': [mu_c1], 'cov': [sigma_c1]}
           },
         {'modes_c0': 1,
          'modes_c1': 1,
          'params_c0': {'a': [2,3], 'b': [4,5]},
          'params_c1': {'a': [1,2], 'b': [7,8]}
           }
]

size = [2700, 300]
```

To make the number of dictionaries and sub-dictionaries one has to create smaller the list values for the distribution parameter keys in the `'params_ci'` sub-dictionaries can either represent individual features or the modes of a feature of a distribution. For example the first two features are distributed according to a multivariate normal distribution where the `'modes_c0'` parameter specifies that the majority class has two modes. Correspondingly the dictionary `'params_c0'` has lists with two parameters as values that represent these modes. For the beta distribution, from which features three and four are sampled, similar lists represent the parameters for feature three (e.g. for class 0: $a = 2, b = 4$) and feature four (again for class 0: $a = 3, b = 5$) respectively.

We realise that this convention makes the dictionaries somewhat of a pain to read, however we deemed it preferable to creating still more sub-dictionaries or lengthening the `dist_parameter_dicts` list, which would have made it still more convoluted to create for higher feature numbers.

Hopefully a description and seeing the code of the generator class makes the structure clear. The initialisation method and the `create_data` method of the generator look as follows:

```
class Multi_Modal_Dist_Generator:

    def __init__(self, distributions, params_dict_list, sizes, random_state = 1234):
        self.sizes = sizes
        self.dists = distributions
        self.params_dict_list = params_dict_list
        self.random_state = random_state


    def create_data(self):

        self.dists_sample_lists = {'c0': [], 'c1': []}

        for l, parameters_dict in enumerate(self.params_dict_list):
```

```
        for i in range(2):

            if (modes:=parameters_dict[f'modes_c{i}']) > 1:

                self.create_multimodal_features(c_id = i,
                                                dist = self.dists[1],
                                                params_dict = parameters_dict[f'params_c{i
}'],
                                                modes = modes,
                                                mixing_weights = parameters_dict[f'
mixing_weights_c{i}'])

            else:
                self.create_unimodal_features(c_id = i,
                                              dist = self.dists[1],
                                              params_dict = parameters_dict[f'params_c{i}'
] )

    self.dists_sample_lists = {key: [array for array in sample_features_list]
                               for key, sample_features_list in self.dists_sample_lists.
items()}

    X_c0 = np.concatenate(self.dists_sample_lists['c0'], axis = 1)
    X_c1 = np.concatenate(self.dists_sample_lists['c1'], axis = 1)

    y_c0 = np.zeros(self.sizes[0])
    y_c1 = np.ones(self.sizes[1])

    self.X = np.concatenate( (X_c0, X_c1), axis = 0)
    self.y = np.concatenate( (y_c0, y_c1), axis = 0)

    # Generate a random permutation of indices
    permuted_indices = np.random.permutation(len(self.X))

    self.X = self.X[permuted_indices]
    self.y = self.y[permuted_indices]
```

Essentially the class takes a list of distributions, the sizes and a list of parameter dictionaries over which the `create_data` method iterates. The following process is done for each class separately, as the modes and number of samples for both classes can be different. For each parameter dictionary the method checks whether the list of distribution parameters it contains represents the parameters of a uni- or multimodal distribution. Depending on whether the `'modes_ci'` parameter is greater than one or not the dictionary is passed to a different function of the class. After the iteration the samples of each class are concatenated along the features dimension, then the samples of both classes are put together and finally they are permuted to avoid a section-wise homogenous dataset.

The actual generation of the samples happens in the functions `create_multimodal_features` and `create_unimodal_features` shown below.

```
def create_multimodal_features(self, c_id, dist, params_dict, modes, mixing_weights):

    size = self.sizes[c_id]

    k = modes

    multinomial = st.multinomial(size, mixing_weights)
    comp_sizes = multinomial.rvs(size = 1)[0]

    acc_list = []

    for i in range(k):

        params = {key: value[i] for key, value in params_dict.items()}

        frozen_dist = dist(**params)

        acc_list.append(frozen_dist.rvs(size = comp_sizes[i]))

    feature_samples = np.concatenate( acc_list, axis = 0)
```

```
        feature_samples = feature_samples.reshape(size, -1)

        self.dists_sample_lists[f'c{c_id}'].append(feature_samples)


    def create_unimodal_features(self, c_id, dist, params_dict):

        size = self.sizes[c_id]

        k = len(next(iter(params_dict.values())))

        sample_features_list = []

        for i in range(k):

            params = {key: value[i] for key, value in params_dict.items()}

            frozen_dist = dist(**params)

            sample_features_list.append(frozen_dist.rvs(size = size))

        sample_features_list = [sample.reshape(size, -1) for sample in sample_features_list]

        self.dists_sample_lists[f'c{c_id}'].extend(sample_features_list)
```

Here the common principle is to extract the parameters of individual features or modes from the lists in the dictionary, create the distribution objects with these parameters, and use their `.rvs` function to generate the samples. The `create_multimodal_features` function is more complex and solves a problem that merits a more detailed description. The common approach to simulate samples from a multimodal distribution is to use mixing weights for the individual components and sample from each component distributions according to this weight. A problem we needed to solve was that multiplying the desired size with the mixing weights did not guarantee an integer number of samples to sample from that component. Rounding up or down would have altered the total number of samples in the mix and made concatenation with the other features impossible. The solution we came up with was to introduce an additional random element and using a multinomial distribution to draw the number of samples that each component would contribute randomly, but with a probability equal to their respective mixing weights. This turned out to be an elegant and functional solution that represents the mixing components fairly accurately, especially for large sample sizes.

In order to use this generator we created a second version of the pipeline. This version differs little from the first approach in principle but has significantly more complicated code as we experimented with the idea of localising iteration, that we had previously done with additional for-loops, to the dedicated classes in the hope of greater efficiency due to less need to pass datasets and recreate instances. So for example, instead of initialising the classifier class for each classifier individually this version initialises the `DictIterClassifier` with all used classifiers at once. As the approach didn't significantly improve computational performance and was superseded by the FMLP approach described in the next chapter we just representatively show the `DictIterClassifier` here.

```
class DictIterClassifier(BaseEstimator, ClassifierMixin):

    def __init__(self, classifiers_dict = {}, classifier_params = {'random_state': 42}):

        classifier_list = [(name, classifier) for name, classifier in classifiers_dict.items()
    ]

        if not isinstance(classifier_params, list):
            self.classifier_params = [classifier_params for _ in range(len(classifier_list))]
        else:
            self.classifier_params = classifier_params

        self.classifier_dict_list = [{'name': name, 'classifier': classifier(**params)}
                                     for (name, classifier), params in zip(classifier_list,
    self.classifier_params)]
```

```
    def fit(self, X, y):

        self.classifiers_dict_list = [
            {**dict, 'classifier': dict['classifier'].fit(X,y)}
            for dict in self.classifier_dict_list
        ]
        return self


    def predict(self, X):

        predictions_dict_list = [
            dict(
                 {key: val for key,val in clsf_dict.items() if key != 'classifier'},
                 **{
                     'predicted_proba': clsf_dict['classifier'].predict_proba(X),
                     'predicted_y': clsf_dict['classifier'].predict(X),
                     'classes': clsf_dict['classifier'].classes_
                    }
            )
            for clsf_dict in self.classifier_dict_list
        ]

        return predictions_dict_list
```

While this pipeline approach didn't reap significant benefits compared to the original version, apart from functioning with the new generator, the elaborate transformations of and iterations over compositions of lists and dictionaries proved very helpful in the design of the FMLP approach.

3.3 Fast Pipeline Approach (FMLP)

Conducting an experiment with the two previous versions of the pipeline involves concatenated for-loop iterations over the respective parameter sets. At each step of an iteration the data is passed between the active class instances of the pipeline and subsequently stored as an instance attribute before transformation. There are two main sources of overhead that these approaches incur due to these operations. For one, every iteration involves the creation and subsequent destruction of large arrays in memory, and while `numpy`'s C implementation guarantees efficient calculations on arrays of fixed size, creation of these arrays incurs significant overhead as large contiguous blocks of memory have to be reserved and released. Secondly, at each step of an iteration the data is passed, i.e. copied, between the individual pipeline components, which again incurs the overhead due to creation and garbage collection.

The key idea of the last pipeline approach is to reduce the impact of these sources of overhead by creating large `numpy` arrays of adequate dimensions once, giving each component of the pipeline a reference to the data instead of a copy, and applying the algorithms directly to sections of these large arrays.

To this end we introduce an intermediate parent class which only has an empty dictionary as a class attribute:

```
class Data():

        data_dict = {}
```

The other classes inherit from `Data` and modify the contents of the `data_dict` dictionary. To direct this process and create the necessary `numpy` arrays we created the `Assessor` class. The `Assessor` takes in the `test_size`, a list of dictionaries for the generator, and as before, a dictionary each for the balancers and classifiers to be used. Its `__init__` method looks like this

```
class Assessor(Data):

    def __init__(self, test_size, generation_dict_list, balancers_dict, classifiers_dict):

        Data.data_dict = {}

        self.test_size = test_size
```

```
        self.generation_dict_list = generation_dict_list

        balancer_list = [(name, balancer) for name, balancer in balancers_dict.items()]

        clsf_list = [(name, classifier) for name, classifier in classifiers_dict.items()]

        self.exp_dim = (len(generation_dict_list), len(balancers_dict), len(classifiers_dict))

        self.data_dict['assignment_dict'] = {(a, b, c): [gen_dict, bal, clsf]
                                             for (a, gen_dict), (b, bal), (c, clsf)
                                             in product(enumerate(generation_dict_list),
                                                        enumerate(balancer_list),
                                                        enumerate(clsf_list)
                                                        )
                                            }
```

Here the baseline dimensions for the `numpy` arrays that are to be created are calculated and saved in `self.exp_dim` and the generation dictionaries and the names and classes corresponding to the balancers and classifiers are stored in the `assignment_dict` under a three dimensional tuple key. This dictionary is essential to assign the methods to the correct positions in the later steps and the correct labels in the output files.

After initialisation the following methods are to be called in sequence if one wishes to execute the pipeline. In the generation function below we first check for the largest number of samples and dimensions to accommodate, then four `numpy` arrays of that size are created and filled with`np.nan` values. Subsequently we iterate over the generation dictionaries in the stored list and assign a generation index to place the generated data in the correct part of the raw data arrays. For each generation dictionary the `FMLP_Generator` class is instantiated and its `prepare_data` method is executed.

```
def generate(self):

    test_size = self.test_size
    table_infos = [extract_table_info(generation_dict) for generation_dict in self.
    generation_dict_list]

    self.d = max([info[0] for info in table_infos])
    n = max([info[1] for info in table_infos])
    a = self.exp_dim[0]

    trainset_size = int((1-test_size)*n)
    testset_size = int(test_size*n)

    self.data_dict['org_X_train'] = np.full(shape = (a, trainset_size, self.d), fill_value =
    np.nan)
    self.data_dict['org_y_train'] = np.full(shape = (a, trainset_size,), fill_value = np.nan)

    self.data_dict['org_X_test'] = np.full(shape = (a, testset_size, self.d), fill_value = np.
    nan)
    self.data_dict['org_y_test'] = np.full(shape = (a, testset_size,), fill_value = np.nan)

    for i, generation_dict in enumerate(self.generation_dict_list):
        generation_dict['gen_index'] = i
        generator = FMLP_Generator(**generation_dict)
        generator.prepare_data(self.test_size)
```

The `FMLP_Generator` doesn't differ much from the generator in the previous section apart from inheriting from `Data` and the `prepare_data` method, which now doesn't return the split raw data but instead inscribes it in the data arrays instantiated earlier.

The next method in line is the `balance` method. It is called with `bal_params_dicts` as a parameter that defaults to an empty dictionary. This parameter is supposed to be a dictionary of dictionaries where each key should be a balancers name and the corresponding value should be a dictionary that contains the instantiation parameters of the balancer, most importantly the `'sampling_strategy'`.

```
    def balance(self, bal_params_dicts = {}):

        a, b, c = self.exp_dim
        y_train = self.data_dict['org_y_train']
```

```
        default_strategy = 'auto'
        max_c1 = max([np.sum(y_train[data_ind] == 1) for data_ind in range(a)])
        max_total_samples = 0

        for data_ind in range(a):
            #check if a balancer parameter dict is given
            if bal_params_dicts:
                #iterate over the parameter dictionaries that are given
                for bal_dict in bal_params_dicts.values():

                    max_total_samples = max(max_total_samples, sum(calculate_no_samples(
    y_train[data_ind], bal_dict['sampling_strategy']).values()))

            #compare to default strategy if not every balancer has a dict
            if len(bal_params_dicts) < b:
                max_total_samples = max(max_total_samples, sum(calculate_no_samples(y_train[
    data_ind], default_strategy).values()))

        k = max_c1 + max_total_samples

        self.data_dict['bal_X_train'] = np.full(shape = (a, b, k, self.d), fill_value = np.nan
    )
        self.data_dict['bal_y_train'] = np.full(shape = (a, b, k, ), fill_value = np.nan)

        data_balancer = FMLP_DataBalancer(bal_params_dicts)

        data_balancer.balance_data()
```

The most complicated part of this step was to find the maximal number of samples for which to reserve space in the balanced data arrays. `Imblearn` balancers allow a wide range of options to set the `'sampling_strategy'` and depending on which strategy the user settles on, the final amount of samples varies drastically. This approach to the pipeline in general sacrifices memory consumption for faster execution time, but optimally the created arrays should not be much larger than they have to be to accommodate the data. To achieve an array that is as small as possible but as large as necessary the `calculate_no_samples` function calculates the number of samples that can be expected depending on the sampling strategy and uses the maximum number among all balancers and datasets to be balanced.

The instantiation of the balancers and the iteration is happening completely in the realm of the `FMLP_DataBalancer` class. This is unlike the generation step where the iteration is in the `generate` method where a new class instance of the generator was created at each iteration. Adding a serialised iteration functionality to the generator would have been considerably more effort to implement as the class already has a high degree of complexity, hence we decided to go for the simpler approach there. For `FMLP_DataBalancer` however it was possible to do the iterations in an elegant way.

Upon initialisation the class creates a sub-dictionary of the `assignment_dict`, dropping the index for the classifier in the keys, and the generator dictionaries and classifier pairs, keeping only name and class of the balancers. The class also stores a dictionary for the parameters of the balancers which is composed of those parameter dictionaries the user specified in the function call and a default dictionary with `'sampling_strategy' : 'auto'` for those balancers without specification.

```
class FMLP_DataBalancer(Data):

    def __init__(self, bal_params_dict = {}):

        self.balancer_dict = {(i,j): assign_list[1] for (i,j,k), assign_list in self.data_dict
    ['assignment_dict'].items()}

        default_dict = {'sampling_strategy': 'auto', 'random_state': 42}
        self.bal_params_dict = {name: bal_params_dict[name]
                                if name in bal_params_dict else default_dict
                                for (name, bal) in self.balancer_dict.values()}


    def balance_data(self):

        X = self.data_dict['org_X_train']
        y = self.data_dict['org_y_train']
```

```
        for (data_ind, bal_ind), (name, balancer) in self.balancer_dict.items():

            X_bal = X[data_ind]
            y_bal = y[data_ind]

            # Drop rows with NaN values
            X_bal = X_bal[~np.isnan(X_bal).all(axis = 1)]
            # Drop columns with NaN values
            X_bal = X_bal[: , ~np.isnan(X_bal).all(axis = 0)]
            y_bal = y_bal[~np.isnan(y_bal)]

            if balancer == None:
                resample = (X_bal,y_bal)

            else:
                balancer = balancer(**self.bal_params_dict[name])
                resample = balancer.fit_resample(X_bal, y_bal)

            n, d = np.shape(resample[0])

            self.data_dict['bal_X_train'][data_ind, bal_ind, :n, :d] = resample[0]
            self.data_dict['bal_y_train'][data_ind, bal_ind, :n] = resample[1]
```

The `balance_data` method retrieves the raw training data from the dedicated arrays and iterates over the reduced tuple indices from the `balancer_dict`. For each combination of balancer and dataset the not-NaN part of the corresponding sub-array of the raw data is selected and new samples are created. The resulting samples are placed in their respective positions in the balancer data array created by the `Assessor`.

The classification and prediction steps follow the same general dynamic, although made somewhat simpler as the appropriate array size is already known. One array is created for the predictions on the test set, another for predicted probabilities on the test set and a final one that stores the class order.

```
    def clsf_pred(self):

        a, n = np.shape(self.data_dict['org_y_test'])

        self.data_dict['clsf_predictions_y'] = np.full(shape = self.exp_dim + (n,), fill_value
     = np.nan)
        self.data_dict['clsf_predictions_proba'] = np.full(shape = self.exp_dim + (n, 2),
    fill_value = np.nan)
        self.data_dict['classes_order'] = np.full(shape = self.exp_dim + (2,), fill_value = np
    .nan)

        data_classifier = FMLP_DataClassifier()

        data_classifier.fit()

        data_classifier.predict()
```

Storing the order is necessary as for a `scikit-learn` classifier the predicted probabilities are ordered by the sequence in which the classes appeared in the training set, which we cannot anticipate due to the randomness in generation. The initialisation method of `FMLP_DataClassifier` follows the same procedure as we already detailed for the `FMLP_DataBalancer`. Also the remaining two methods follow similar steps. In both the `fit` and the `predict` method the iteration proceeds over a transformed version of the `assignment_dict` where only the classifier entries were selected and their classes instantiated.

```
class FMLP_DataClassifier(Data):

    def __init__(self, clsf_params_dict = {}):

        default_dict = {'random_state': 42}
        classifier_dict = {key: assign_list[2] for key, assign_list in self.data_dict['
    assignment_dict'].items()}

        classifier_dict = {key: (name, clsf(**clsf_params_dict[name]))
                           if name in clsf_params_dict else (name, clsf(**default_dict))
                           for key, (name, clsf) in classifier_dict.items()}
```

```
        self.classifier_dict = classifier_dict


    def fit(self):

        X = self.data_dict['bal_X_train']
        y = self.data_dict['bal_y_train']

        for (i,j,k), (name, clsf) in self.classifier_dict.items():

            X_fit = X[i, j, :, :]
            y_fit = y[i, j, :]

            # Drop rows with NaN values
            X_fit = X_fit[~np.isnan(X_fit).all(axis = 1)]
            # Drop columns with NaN values
            X_fit = X_fit[: , ~np.isnan(X_fit).all(axis = 0)]

            y_fit = y_fit[~np.isnan(y_fit)]

            self.classifier_dict[(i,j,k)] = (name, clsf.fit(X_fit, y_fit))

        return self


    def predict(self):

        X = self.data_dict['org_X_test']

        for (i,j,k), (name, clsf) in self.classifier_dict.items():

            X_test = X[i, :, :]

            # Drop rows with NaN values
            X_test = X_test[~np.isnan(X_test).all(axis = 1)]
            # Drop columns with NaN values
            X_test = X_test[: , ~np.isnan(X_test).all(axis = 0)]

            n_i = len(X_test)

            self.data_dict['clsf_predictions_y'][i, j, k, : n_i] = clsf.predict(X_test)
            self.data_dict['clsf_predictions_proba'][i, j, k, : n_i, :] = clsf.predict_proba(
X_test)
            self.data_dict['classes_order'][i, j, k, :] = clsf.classes_
```

Every dataset-balancer-classifier combination in this tensor is then first `fit` with the balanced data, corresponding to the position in the tensor, and then called on to give predictions in `predict`. Each time we select a view of the whole data we drop the NaN components from the view before passing it.

The next step to achieve a similar functionality as the previous pipeline versions was to calculate the standard metrics. Once one has understood the design of the previous functions in `Assessor` this one is straightforward to understand due to the similarity. The user passes a dictionary of named standard metric-functions. 'Standard' here essentially refers both to the commonality of the metrics as well as the technical aspect of their calculation: Only the original test sets and the trained classifiers predictions on them are necessary. Should the user not pass a dictionary with the desired standard metrics a default will be used. We applied this default to the majority of our tests.

```
    def calc_std_metrics(self, std_metrics_dict = {}):

        default_metrics = {
            'accuracy': accuracy_score,
            'precision': precision_score,
            'recall': recall_score,
            'F1 score': f1_score,
            'ROC AUC Score': roc_auc_score,
        }

        metrics_dict = std_metrics_dict or default_metrics
```

```
        self.data_dict['std_metrics_res'] = np.full(shape = self.exp_dim + (len(metrics_dict)
    ,), fill_value = np.nan)

        metrics = FMLP_Metrics(metrics_dict)

        metrics.confusion_metrics()

        std_metrics_res = self.data_dict['std_metrics_res'].reshape(-1, len(metrics_dict))

        results_df = pd.DataFrame(std_metrics_res, columns= [name for (name, metr_func) in
    metrics.std_metric_list])

        reference_list = [self.data_dict['assignment_dict'][(i, j, k)]
                          for i in range(self.exp_dim[0])
                          for j in range(self.exp_dim[1])
                          for k in range(self.exp_dim[2])]

        reference_list = [extract_table_info(alist[0])+[alist[1][0], alist[2][0]] for alist in
     reference_list]

        reference_df = pd.DataFrame(reference_list, columns= ['n_features',
                                                              'n_samples',
                                                              'class_ratio',
                                                              'distributions',
                                                              'balancer',
                                                              'classifier'])

        results_df = pd.concat([reference_df, results_df], axis = 1)

        return results_df
```

What is different in this part of the pipeline is that the names and the crude information from the generator dictionaries contained in the `assignment_dict` is transformed into a reference list that we use to create the `results_df`. The `FMLP_Metrics` classes `confusion_metrics` function is so similar in structure to the functions previously described that we will just show the code without additional explanation:

```
def confusion_metrics(self):

        y_test = self.data_dict['org_y_test']
        y_pred = self.data_dict['clsf_predictions_y']

        for (i,j,k) in self.data_dict['assignment_dict']:

            y_i_test = y_test[i]
            y_i_test = y_i_test[~np.isnan(y_i_test)]

            y_clsf_pred = y_pred[i, j, k]
            y_clsf_pred = y_clsf_pred[~np.isnan(y_clsf_pred)]

            evaluation = np.array([metr_func(y_i_test, y_clsf_pred) for (name, metr_func) in
    self.std_metric_list])

            self.data_dict['std_metrics_res'][i, j, k, :] = evaluation
```

3.4 Discussion

The greatest challenge overall in the creation of these pipelines was the administration of parameters. Allowing all segments of the pipeline to be general enough to accept the large variety of parameters that generator, balancers and classifiers possess, while maintaining usability and a streamlined workflow, was very difficult. We wanted the pipeline to be as general as possible to enable a large body of experiments to be conductable but assure that a user does not have to provide all the parameters in a short experiment.

As mentioned before the FMLP approach will on average be faster but as it stores large arrays in memory that remain throughout the execution of the pipeline, it places a much heavier burden on

available RAM memory. While we did not have the time to implement this here, the computational efficiency of this approach could be greatly increased still with the use of parallel processing and GPU recruitment. Furthermore one could create custom versions of the `imblearn` balancing algorithms and `scikit-learn` classifiers that sacrifice some generality in favour of efficient serialisation that makes greater use of the optimised matrix operations of modern GPUs. The FMLP approach allows to test various balancers and classifiers with the freedom to set all parameters for each of them. It does not however enable serialised experiments where different parameter sets are passed to the same balancer or classifier without additional exterior iteration.

3.5 Hypothesis testing

The **Hypothesis-T-Test** class is designed to facilitate hypothesis testing, specifically t-tests, on datasets stored in CSV format. It enables us to compare the effectiveness of different combinations of data balancing and classification methods in terms of key performance measures. The t-test is an appropriate choice in this case, where we want to apply multiple pairwise comparisons of continuous variables (the target measures). Hereby is an overview of its functionality.

The class constructor accepts several essential parameters:

- **target list**: a list of target performance measures;
- **column bal**: the name of the file column addressing the data balancing method
- **column clas**: the name of the file column addressing the classifier method
- **test combination balancers**: a list of tuples, each containing a combination of two balancing methods for comparison
- **test combination classifiers**: a list of tuples, each containing a combination of two classifier methods for comparison
- **alpha**: the test significance level, set by default to 0.05, a typical choice in hypothesis testing.

The core functionality of the class resides in the **perform-t-test** method, which consists of the following steps:

- Initialization of an empty list, **results**, to store the outcomes of the t-tests.
- For each target performance measure, the class iterates through the provided combinations of balancing methods and classifier methods.
- The data is filtered based on the selected balancing or classifier method, creating two distinct groups.
- A t-test is conducted on the two groups, producing statistical values such as the t-statistic and p-value.
- The mean values of the target performance measure are calculated for each group.
- The test results, including the target performance measure, compared methods, mean values, t-statistic, and p-value, are organized into dictionaries and appended to the **results** list.
- After completing all t-tests, the results are compiled into a pandas DataFrame and saved to a CSV file. This file serves as a comprehensive record of the statistical comparisons.

3.6 Linear regression Analysis

Another crucial step in our pipeline is to thoroughly explore the relationships between our target measures and the parameters of interest that we allowed to vary in the preceding stages of our project. To achieve this, we've developed the **Linear Regression Analysis** class. This class empowers us to conduct in-depth investigations into how specific predictor variables influence our chosen target metrics. In the following, we will delve into the functionality of this class and showcase its applications in our data analysis process. After loading a dataset from a specified CSV file and storing it in a pandas DataFrame, the data need to be set up for regression analysis. This is done through the **prepare data** method, which takes two sets of input variables: categorical and continuous regressors. The categorical variables are encoded into binary format and combined with the continuous regressors to create the input matrix **X**. The actual analysis is then implemented in the function **perform linear regression**, which takes the following inputs:

- **target**: the target metric that we want to predict
- **regressors**: a list of predictor variables used in the regression analysis

The function performs the following steps:

- Splits the dataset into training and testing sets.
- Fits a linear regression model to the training data.
- Predicts the target metric for the test data.
- Calculates various regression metrics, including mean squared error, mean absolute error, coefficient values, coefficient of determination (R-squared), F-value, and p-value.
- Stores the results in a pandas DataFrame and returns it.

Finally, the method **plot target vs regressors** repeats the analysis and loops through each regressor in the regressors list. For each regressor, the corresponding predicted values **y values** are calculated as a sum of the intercept and the product between the regression coefficients and the data. Next, a scatter plot is produced to graphically observe the relationship.

References

[1] Nitesh, V., Chawla., Kevin, W., Bowyer., Lawrence, O., Hall., W., Philip, Kegelmeyer. (2002). SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 16(1):321-357. doi: 10.1613/JAIR.953

[2] Haibo He, Yang Bai, E. A. Garcia and Shutao Li, "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), Hong Kong, 2008, pp. 1322-1328, doi: 10.1109/IJCNN.2008.4633969.