# CLL-788

# Process Data Analytics

# Term Paper Report

- ## Abstract:

With the "BIKED" dataset—which includes 4500 unique bicycle designs from a wide variety of designers—into consideration, this report seeks to reproduce the results of that study. For data-driven design applications and method development, the dataset offers a treasure trove of design information, such as images of assemblies and components, numerical parameters, and class labels. We want to replicate the paper's findings using multi-class classification code in Python. Here we'll get into some of the main concerns raised by the original paper, including how to map out the bike design space, how machines comprehend bike designs, and how to train Variational Autoencoders using BIKED data to come up with new ideas. We hope that by doing this, we can prove that the dataset is useful and help data-driven design techniques progress.

- ## Overview:

Processing and classifying the BIKED dataset is done in a methodical way by the code that is implemented. At first, the dataset is cleaned up by separating the labels and features. Then, the categorical features are encoded using one-hot encoding. After that, to make sure that the feature scales are consistent, the dataset is normalised using Min-max normalisation. The given script 'getXML.py' is used to visualise example bike designs, looking like Fig. 2 from the BIKED publication. Figure 7 from the BIKED study shows the normalised dataset being subjected to dimensionality reduction techniques such as Principal Component Analysis (PCA) and t-distributed Stochastic Neighbour Embedding (t-SNE). The next step is multiclass classification, which involves training a classifier based on neural networks to divide the information into nineteen distinct bike classes. At last, the classifier's efficacy is assessed by creating a confusion matrix. The code tries to improve upon and repeat the results of the original study by including data preprocessing, visualisation, dimensionality reduction, and classification tasks.

- **<u>Explanation of the code:</u>**

Firstly, the code undertakes extensive data preprocessing steps to ready the dataset for analysis and modeling. Initially, it imports NumPy and Pandas, then loads two datasets, "BIKED_datatypes.csv" and "BIKED_reduced.csv", into DataFrames data_df and reduced_data, respectively. The target variable 'BIKESTYLE' is separated into variable Y. Redundant 'Unnamed: 0' columns are removed from reduced_data. Missing values in categorical columns are filled with the mode, while those in numeric columns are replaced with the mean. Boolean columns' missing values are filled with the mode. One-hot encoding is applied to categorical columns, and redundant columns are removed. Boolean values are converted to numerical equivalents. Min-max scaling standardizes numeric features. The transformed dataset, X_scaled, is printed for review, completing the crucial preprocessing steps to prepare the dataset for analysis and modeling.

Then the code implements a function `genBCAD` that takes a DataFrame of standardized features (`df`), a source file path (`sourcepath`), and a target directory path (`targetpath`). It iterates over a subset of rows (`num`) in the DataFrame, representing different bike models, and reads a template BikeCAD file (`sourcefile`) line by line. For each line, it identifies parameters enclosed within XML tags and checks if they match column labels in the DataFrame. If a match is found, it writes the corresponding value from the DataFrame to the target BikeCAD file (`targetfile`), formatting it appropriately based on data type. Special considerations are made for boolean values and floats to ensure compatibility with BikeCAD requirements. The resulting BikeCAD files are saved in the specified directory. The function also includes a helper method `find_between` to extract substrings between two specified strings. Overall, this code facilitates the automated generation of customized BikeCAD files based on input feature data, streamlining the process of creating bike designs for analysis and visualization.

Then the code performs dimensionality reduction using Principal Component Analysis (PCA) to visualize high-dimensional data in two dimensions. Initially, the standardized data (`standard_data`) is prepared by scaling the input features. Then, the covariance matrix (`covar_matrix`) is computed by multiplying the transposed data matrix with itself. Next, the eigenvalues and eigenvectors of the covariance matrix are calculated using the `eigh` function from SciPy's linear algebra module, retaining only the top two eigenvalues and corresponding eigenvectors. The resulting eigenvectors are transformed and used to project the original data onto a lower-dimensional space, yielding `new_coordinates`. These new coordinates are then visualized in a scatter plot using Matplotlib and Seaborn, facilitating the exploration of the dataset's structure and potential clustering patterns. Finally, a DataFrame (`matrix_df`) is created to organize the projected data, and a scatter plot is generated to visualize the principal components. This process provides insights into the intrinsic structure of the dataset while reducing its dimensionality for easier interpretation and visualization.

The code then utilizes Principal Component Analysis (PCA) from scikit-learn for dimensionality reduction without visualization. Initially, a PCA object is instantiated, and the number of principal components is set to 400. The `fit_transform` method is then applied to the standardized data (`sample_data`), resulting in `pca_data`, which represents the original data projected onto the principal components. Next, the percentage of explained variance for each principal component is calculated and cumulated to obtain the cumulative explained variance (`cum_var_explained`). This cumulative explained variance is plotted against the number of components to visualize the proportion of variance retained with increasing dimensions. Finally, the plot illustrates that approximately 90% of the variance can be explained with 200 dimensions, offering insight into the trade-off between dimensionality reduction and retained variance.

Following, the code primarily focuses on dimensionality reduction and visualization techniques, specifically Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE). Initially, PCA is applied to reduce the dimensionality of the data to two dimensions (`n_components = 2`). The `fit_transform` method transforms the standardized data (`sample_data`) into a new dataset (`pca_data`) containing only two principal components. The shape of `pca_data` is printed to verify the dimensionality reduction. Subsequently, the transformed PCA data is organized into a DataFrame (`pca_df`) for visualization, where the two principal components are plotted against each other using Seaborn and Matplotlib.

Following PCA, t-SNE is employed for further dimensionality reduction and visualization. Initially, a subset of the standardized data (`data_1000`) is selected for processing. The t-SNE model is constructed with default parameters and fitted to the data, producing a two-dimensional representation (`tsne_data`). Similar to PCA, the t-SNE data is structured into a DataFrame (`tsne_df`) for visualization, and the scatter plot is generated. The process is repeated with a modified perplexity value (`perplexity=50`) to observe its effect on the visualization. Additionally, the t-SNE model is built using the entire dataset (`data_5k`) for a more comprehensive analysis, generating visualizations to explore the dataset's structure. Finally, a t-SNE model is constructed with specific parameters (`perplexity=40, n_iter=500`) using the complete training dataset, and its resulting visualization is produced. Overall, this code segment illustrates the application of PCA and t-SNE for dimensionality reduction and visualization to gain insights into high-dimensional datasets.

The code then serves to set up the environment and define essential functions and variables for subsequent operations. Initially, necessary libraries such as NumPy, pandas, Matplotlib, SciPy's optimization module, and MATLAB data loading functions are imported. The `sigmoid` function is defined to implement the sigmoid activation function, which is fundamental in logistic regression and neural networks. A list `class_names` is created to store the names of different bike styles. The data type of `Y` is printed to inspect its type, followed by its conversion to a NumPy array to facilitate further operations. Each class label in `Y` is then mapped to its corresponding index in the `class_names` list, effectively encoding categorical labels into numerical values. Next, variables are initialized to specify the sizes of the input layer, hidden layers, and the number of output labels, essential parameters for defining the neural network architecture. Overall, this part of the code establishes the necessary functions and variables required for subsequent classification tasks, particularly for neural network training and evaluation.

The `sigmoidGradient` function computes the gradient of the sigmoid function evaluated at a given input `z`. It first calculates the sigmoid function of `z` using the formula $1/(1+e^{-z})$. Then, it computes the gradient of the sigmoid function using the derivative formula
sigma(z) *(1 – sigma(z)), where sigma(z) is the sigmoid function. The resulting gradient `g` has the same shape as the input `z` and represents the rate of change of the sigmoid function with respect to the input. This function is essential for backpropagation in neural networks, where it helps compute the error derivatives with respect to the network's parameters during the training process.

The `randInitializeWeights` function randomly initializes the weights of a layer in a neural network. It takes two parameters, `L_in` and `L_out`, representing the number of incoming and outgoing connections, respectively. Additionally, it accepts an optional parameter `epsilon_init`, which specifies the range of values the weights can take from a uniform distribution. Inside the function, weights `W` are randomly initialized using the formula **$W=rand((L_{out}*(1-L_{in}))*2*epsilon_{init} - epsilon_{init})$** where `rand` generates random values between 0 and 1. These initialized weights are crucial for the effective training of neural networks, ensuring that the network learns diverse features and avoids getting stuck in local minima during optimization.

Finally, the `nnCostFunction` function implements the cost function and gradient computation for a neural network with multiple layers. It takes several parameters, including the neural network parameters `nn_params`, the sizes of different layers, input data `X`, labels `y`, and a regularization parameter `lambda_`. Inside the function, the neural network parameters are reshaped into weight matrices `w1`, `w2`, `w3`, and `w4` corresponding to the layers of the network. The forward propagation computes activations through each layer using the sigmoid activation function. Then, the cost function is calculated using the cross-entropy loss with regularization to prevent overfitting. The backpropagation algorithm computes gradients of the cost function with respect to the parameters of each layer, considering the regularization term. Finally, the gradients are unrolled into a single vector and returned along with the cost value. This function is vital for training neural networks using gradient-based optimization algorithms like gradient descent.

The last section of the code performs the training of a neural network using optimization techniques. First, a regularization parameter `lambda_` is set to 0.1 to control overfitting during training. Then, a shorthand notation is created for the cost function to be minimized using lambda functions. The `costFunction` lambda function encapsulates the `nnCostFunction_vectorized` function, which computes the cost and gradients of the neural network given the parameters `p`, input data `X`, labels `Y`, and the regularization parameter `lambda_`. Next, an optimization algorithm is employed to minimize the cost function. The `optimize.minimize` function is called with arguments including the cost function, initial neural network parameters `initial_nn_params`, and optimization options such as using the TNC method and setting a maximum number of function evaluations (`maxfun`). After optimization, the resulting parameters `nn_params` are obtained from the optimization result. These parameters are then reshaped to obtain weight matrices `w1`, `w2`, `w3`, and `w4` for each layer of the neural network. Finally, the `predict` function is defined to make predictions using the trained neural network parameters on the input data `X`, and the accuracy of the predictions is computed and printed. This process allows for training a neural network model and using it to make predictions on new data.