MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIONES

MACHINE LEARNING LAB

FINAL REPORT

Methodology and Tools for Developing Deep Learning Models

Case Study: Collision Avoidance Challenge

Sergio Pérez Morillo
Jaime Pérez Sánchez
Carlos Andrés Ramiro

Course 2019/20

Table of Contents

1. Environment for Experiments

In this section we will talk about the equipment we have used to carry out this project, we will deal with both hardware and software elements.

1.1. Hardware Equipment

The hardware architecture utilized to test this methodology consists of:

- A laptop that includes an Intel Core i7-8750H CPU, a 16GB DDR4 RAM, and Kubuntu as the operating system.
- A laptop that includes an Intel Core i5-6300HQ @3.2GHz CPU, a 16GB RAM and Arch Linux as the operating system.
- A laptop computer that has an Intel Core i5-5200 @2.2 GHz CPU, a 4GB RAM and Windows 10 as the operating system.
- We also used the Google Colab platform for doing some tasks that would have taken much time with our personal computers.

1.2. Software Tools

It has been decided that this project would be developed in **Python** (version 3.7) programming language. The main reasons for this choice are previous experience with the language from the developers, simple syntax, a large number of libraries for Machine Learning and Deep Learning development (with smoothly difficult curves) and simplicity of integration in case of model deployment. The software architecture consists of several Jupyter notebooks running on Anacondas' virtual environments that feature several Python libraries for different tasks. In the following, we will briefly describe the Python libraries used during the project.

- **TensorFlow:** End-to-end Deep Learning, dataflow and differentiable programming platform. It is one of the most important libraries for the development of Neural Networks models, developed by Google. In this work, we have used version 2.0, which already includes the high-level library Keras in it.
- **Keras:** High-level library to enable fast, modular and user-friendly development of Neural Network models. In version 2.0 of TensorFlow, this library is already included.
- **Keras-tuner:** Framework for Hyperparameter Optimization on Keras. Includes searching methods based on grid-search, gradient-based, random search, and Bayesian optimization among others.

- **NumPy:** Fundamental package for scientific computing, including powerful N-Dimensional array objects and functions (essential for Machine Learning).

- **Pandas:** Data structures management and data analysis library. It provides a high-performance and easy-to-use framework to manage and manipulate datasets.

- **Scikit-learn:** Fundamental Machine Learning library that includes a large number of algorithms (classification, regression, clustering...) together with preprocessing and evaluation functions.

- **Matplotlib:** 2D plotting library which produces publications quality figures in a variety of hard copy formats and interactive environments across platforms.

- **Seaborn:** Data visualization library over Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.
- **Featuretools:** Library for automatic features engineering from temporal and relational datasets.

- **Category-encoders:** Set of Scikit-learn-style transformers for encoding categorical variables into numeric with different techniques.
- **Impyute:** Library for easy implementation of missing data imputation algorithms.
- **Imbalanced-learn:** Library for easy implementation of re-sampling techniques. Includes methods of under-sampling, over-sampling and combinations of both.
- **DeepStack:** Library built on top of Keras for creating ensembles with Deep Learning models. Includes stacking and Dirichlet Markov ensemble methods.

2. Data Manipulation

In this section, we discuss the preliminary data wrangling to transform the dataset into usable arrays for later steps. Bear in mind that the procedure has been simplified for a better understanding as some parts are a bit confusing. This is because the data was really raw and need lots of specific processing to be useful. For the full implementation, please visit our Github accounts **[1][2][3]**, the code is partially commented.

First, we explain the main reshape transformations and padding procedures. Then, we talk about imputation strategies. Finally, we see oversampling and undersampling methods. In this report, the procedure is to describe step-by-step the implementation of relevant parts alongside a code (or pseudocode) snippet.

2.1. Dataset Versions & Sequence Padding

As explained in the PRDL report, there are two main reshape transformations, one for creating a 3D input array and another one for a 2D input array (the “X” array). Both use the same target feature “y”. The Python libraries used to transform the data are **Numpy** and **Pandas**.

The following procedure is valid for our two versions. The first part is to import the data and declare some variables such as timestep (window size), padding value, and the X and y scalers (this will be explained in section 6, but it is important to be aware that data must be scaled before padding it to vastly ease the this process).

```python
df = pd.read_csv("train_data.csv")
timesteps = 17 # Sequence length
pad_value = -1
X_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()
```

Then, in the most naive approach to imputation, we remove the missing values to avoid further problems. After that, we filter the events that not meet the two conditions explain in the PRDL report:

```python
# Filtering events with min_tca > 2 or max_tca < 2
def conditions(event):
    x = event["time_to_tca"].values
    return ((x.min()<2.0) & (x.max()>2.0))
df = df.groupby('event_id').filter(conditions)
```

Note that we use numpy operations to compute the conditions as they are much faster than pandas or built-in Python operations and *groupby* operations consume a lot of time. Then, we get the ‘y’ array and the ‘X’ (still as dataframe) while scaling them (this was omitted). Note the use of the feature ‘event_id’ to group the data as it is the identifier of the events.

```python
# Getting y as 1D-array (tca_min of each event)
y = df.groupby(["event_id"])["risk"]
      .apply(lambda x: x.iloc[-1])
# Getting X as 2D-df (dropping rows with tca < 2)
df = df.loc[df["time_to_tca"]>2]
```

After scaling the data and getting y and a preliminary version of X, we shape the latter into a 3D-array (for both dataset versions). This part is a little bit tricky. First, we initialize the 3D array and pad it with the chosen padding value (so events that have a smaller sequence size than the chosen timestep will be already padded). Then, we created a function to apply after an event *groupby* in which we fill the array row event by event (row by row) looking first at their size to not get an out of bound error. The implementation is as follows:

```python
# Initializing X array and padding it
X = np.zeros((events,timesteps,features))
X.fill(pad_value)
i = 0

def df_to_3d_array(event):
    global X, i
    # Reshaping event into a 3D sample (1, timesteps, features)
    row = event.values.reshape(1,event.shape[0],event.shape[1])
    # is the selected sequence length (timesteps) longer
    # than the real sequence length (event sequence)?
    if(timesteps>=row.shape[1]):
        X[i:i+1,-row.shape[1]:,:] = row
    else:
        X[i:i+1,:,:] = row[:,-timestep:,:]
    # Index to iterate over X array
    i += 1
    # Dataframe remains intact, while X array has been filled.
    return event

df.groupby("event_id").apply(df_to_3d_array)
```

After this operation, we get out X, y arrays and their corresponding scalers fitted. So far, this procedure is applied to both versions of the dataset. But as mentioned before, the second version has two dimensions, so it needs an extra and simple step.

It needs simply to be reshaped into a 2D array to have the desired and time-shifted features. Finally, we create an array of the names of the features (e.g. “risk_t-3” refers to the risk feature shifted three times) to utilize it as an input to create also an X data frame and ease its use in data analyses of feature steps. The implementation of this extra step is the following:

```
# Reshaping to a 2D array
X = X.reshape(X.shape[0], timestep*X.shape[2])
# Naming time-shifted features
shifted_columns = []
original_columns = list(df.columns)

for i in range(timestep-1,-1,-1): # backward iteration
    for column in original_columns:
        shifted_columns.append(column+"_t-"+str(i))

# Creating df from X and the names
X = pd.DataFrame(X, columns=shifted_columns)
```

2.2. Imputation Techniques

As explained in the PRDL report, the first approach (naive) was to eliminate all rows containing any missing value in the form of NaN. To do this, the *dropna* function of the *Pandas* library has been used.

```
X = X.dropna(axis=0, how='any')
```

The parameter *axis=0* of the function indicates that what we are going to eliminate are the rows with missing values. If we want to remove all the columns with missing values we have to set *axis=1*. The second approach and the one finally used, was to use a data imputation method based on the k-NN algorithm, implemented in the *Impyute* library.

```
from impyute.imputation.cs import fast_knn
columns = list(X.columns.values)
# Input the missing values with k-nn algorithm. Output = numpy array
imputed_training = fast_knn(X.values, k=30)
# Save the obtained array with the corresponding names of the columns
X = pd.DataFrame(imputed_training, columns=columns)
```

This method is very computationally expensive, becoming impossible to execute in our personal computers when it was carried out on the complete dataset. So it was executed most of the times on the Google Colab platform.

2.3. Oversampling & Undersampling

To address the problem of unbalanced classes in the dataset, several algorithms included in the Imbalanced-learn library were implemented. The algorithm that offered the best results was SMOTE (Synthetic Minority Over-sampling Technique).

```
from imblearn.over_sampling import SMOTE
sm = SMOTE(random_state=42, k_neighbors=58)
X_train, y_train = sm.fit_resample(X_train, y_train))
```

As we can see in the following figures, with the use of this method it was possible to completely balance the classes of the training targets. The validation and test classes must be kept unbalanced since this is the data we use to validate the model and we want to predict, respectively.

```
High Risk: 66
Low Risk: 6207
Percent High Risk: 1.052128168340507
```

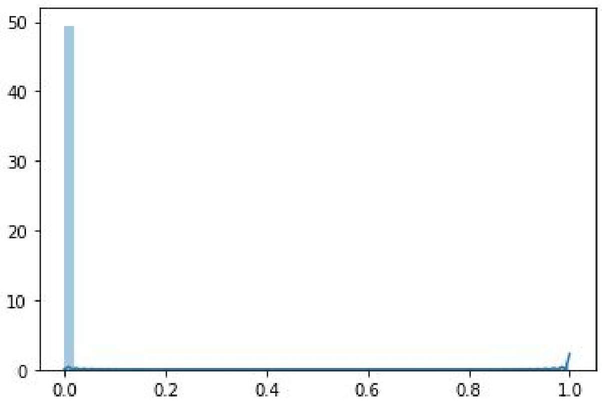


Figure 1 - Distribution of classes in the training set (before applying over-sampling)

```
High Risk: 6207
Low Risk: 6207
Percent High Risk: 50.0
```

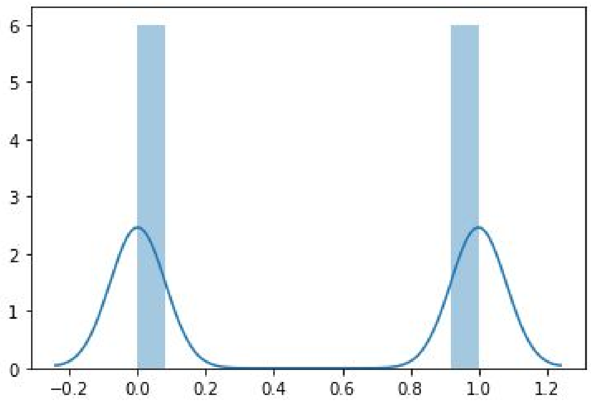


Figure 2 - Distribution of classes in the training set (after applying over-sampling)

3. Exploratory Data Analysis

The next step is to perform data analysis to gain insights from the different features in our processed dataset. To do this, we have used different libraries to visualize the data and thus be able to interpret them more easily.

We have visualized scatters, distributions, and correlations from categorical features, numerical features, and the combination of both. We have used the libraries *Matplotlib* and *Seaborn* since they provide a user-friendly framework and very useful visualization methods.

3.1. Distributions & Correlations

Our first task would be to observe the distributions of the numerical variables. However, we had a dataset with a large number of features so it was very complicated and laborious to evaluate the distribution of each of them. That is why we will analyze the variables that have the highest correlation with risk.

We observe the distributions of the variables and how they relate to the rest of the variables using the *pairplot* function from *Seaborn*:

```
sns.set(style="ticks")
fig = sns.pairplot(df,palette="Set3")
axes = fig.axes.flatten()
for ax in axes:
    ax.set_xlabel(ax.get_xlabel(),fontsize=30)
    ax.set_ylabel(ax.get_ylabel(),fontsize=30)
plt.plot()
```

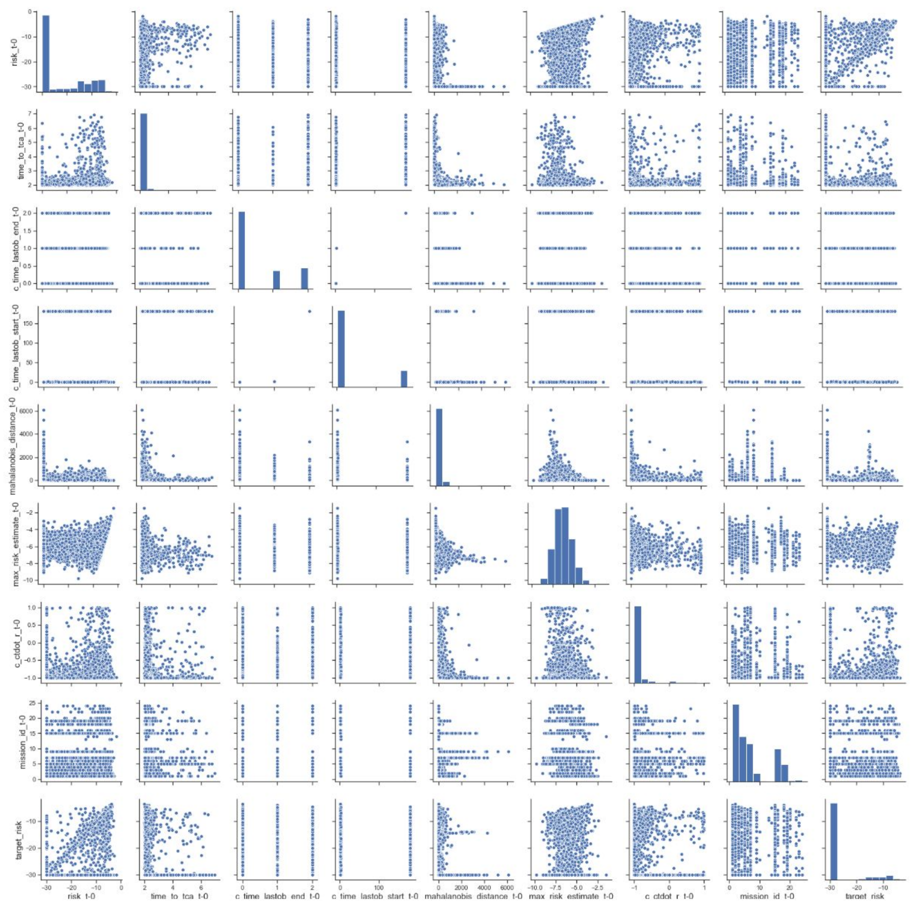


Figure 3. - Scatter Plot

Then I plotted a heatmap to check numerically the correlation between features.

```
plt.figure(figsize=(8,8))
hm = sns.heatmap(df.corr(),cmap="RdYlGn", annot=False)
bottom, top = hm.get_ylim()
hm.set_ylim(bottom + 0.5, top - 0.5)
hm.set_xticklabels(hm.get_xmajorticklabels(), fontsize = 15)
hm.set_yticklabels(hm.get_ymajorticklabels(), fontsize = 15)
plt.show()
```

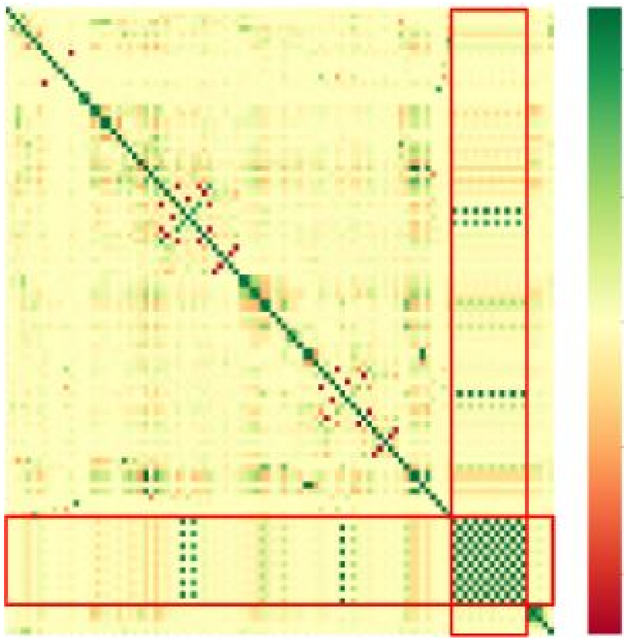


Figure 4. - Correlation Matrix

As for the categorical variables we made a bar_plot because of its simplicity. We wrote a simply loop to show them all at once.

```
for col in df.select_dtypes(include='category').columns:
        fig = sns.catplot(x=col, kind="count", data=df, palette="Set3")
        axes = fig.axes.flatten()
        axes[0].set_xlabel(col,fontsize=20)
        axes[0].set_ylabel(" ",fontsize=20)
        axes[0].set_xticklabels(axes[0].get_xmajorticklabels(), rotation=90,
fontsize = 13)
        axes[0].set_yticklabels(axes[0].get_ymajorticklabels(), fontsize =
13)
        plt.show()
```

The following figure shows an example of one of the resulting plots:

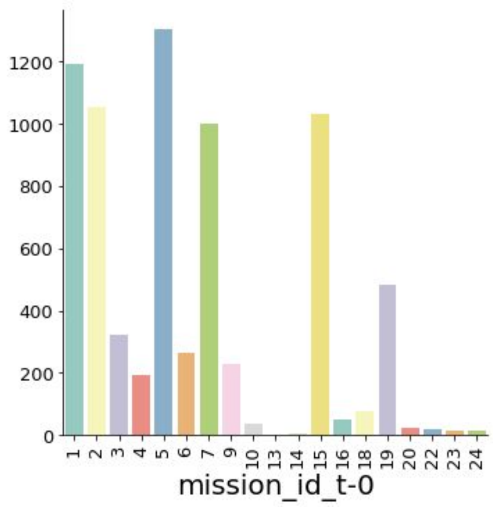


Figure 5. - Categorical Feature distribution

Finally we combined both categorical and numerical features to better understand them and thus draw more insightful conclusions. To do that, we plotted boxplots of each numerical feature as a function of the object type and the mission id.

```
for col in df.select_dtypes(include='float64').columns:
    sns.boxplot(x="c_object_type_t-0", y=col, data=df, palette="Set3")
    plt.show()
```

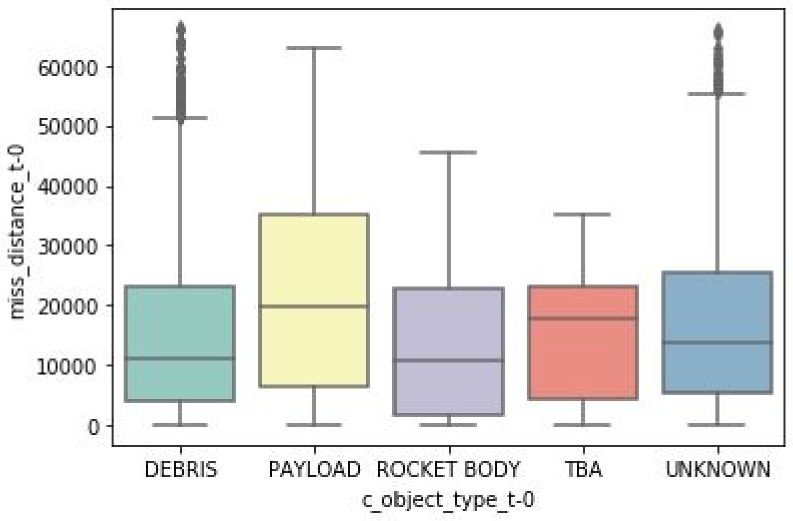


Figure 6. - Boxplot comparing features

Then we plotted boxplots of the target feature "risk" as a function of each categorical feature:

```
for col in df.select_dtypes(include='category').columns:
    hm = sns.boxplot(x=col, y="risk_t-0", data=df, palette="Set3")
    hm.set_xticklabels(hm.get_xmajorticklabels(),rotation=90,fontsize = 13)
    hm.set_yticklabels(hm.get_ymajorticklabels(), fontsize = 18)
    hm.set_xlabel(hm.get_xlabel(),fontsize=20)
    hm.set_ylabel(hm.get_ylabel(),fontsize=20)
    plt.show()
```

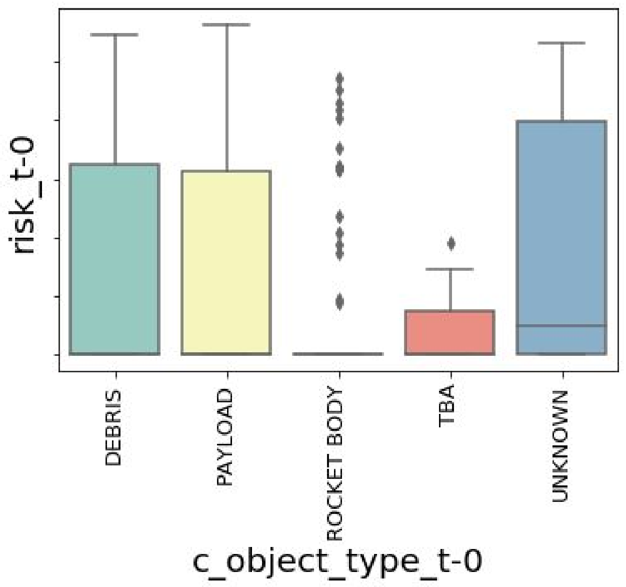


Figure 7. - Categorical features with target feature plot

3.2. Dimensionality Reduction

For implementing and computing the dimensionality reduction technique PCA, we used **Scikit-learn** and **pandas**. First, we selected the second dataset version (X is a 2D-array with time-shifted features) that is already scaled (critical from PCA). Then, we initialized a PCA of three components, fitted the data, and stored it in a data frame (mandatory to plot anything in Plotly express) alongside the boolean target feature 'y' to use it as hue.

```
# Initializing the PCA and fitting the data
pca = PCA(3)
X_pca = pca.fit_transform(X) # already scaled!

#Storing the data into a df
df = pd.DataFrame({ 'pca-one': X_pca[:,0], 'pca-two': X_pca[:,1],
                    'pca-three': X_pca[:,2], 'anomaly': y_boolean })
```

On the other hand, for plotting the 3D scatter we used **Plotly express**. This library is both powerful and really easy to use. Simply by using one of its functions, *scatter_3d* we created an interactive 3D scatter plot. This enables us to assess the results with ease. We also generate a 2D scatter plot with *seaborn*, but we did not include it as it shows worse results and seems redundant. The pseudocode of the 3D plot and the results are the following:

```
# Plotly 3D scatter of PCA results
fig = px.scatter_3d(df_pca, x='pca-one', y='pca-two',
                    z='pca-three', color='anomaly' )
fig.show()
```

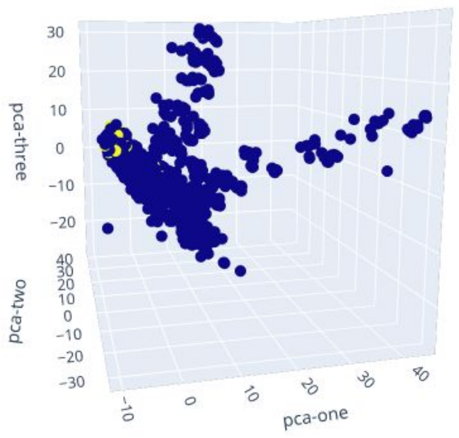


Figure 8. - 3D scatter plot of PCA results.

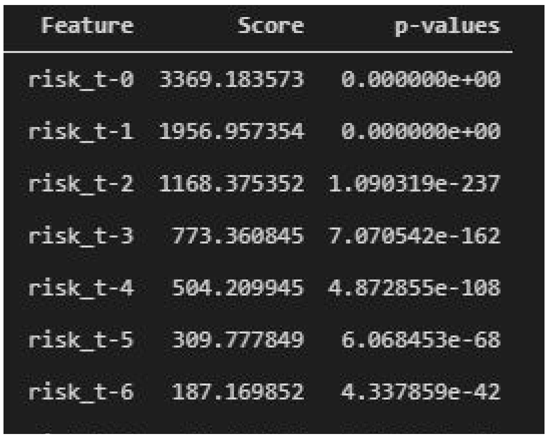
4. Feature Selection

In this section we will discuss the techniques used for the development of feature selection, which will help us reduce the dimensions of the dataset by focusing on the most important variables for predicting risk.

4.1. Filter Methods: Univariate Selection

For performing univariate selection we used Sklearn's functions SelectKBest and f_regression and plot the results again with seaborn. First, we fitted the SelectKBest model with the f_regression function, then we created a dataframe to store the results and plot them. The output is the following:

```
score_func = f_regression
k = 3
show_df = 100
show_plot = 20

#Applying SelectKBest to extract the 10 best features
bestfeatures = SelectKBest(score_func=score_func, k=k)
fit = bestfeatures.fit(X,y)
...
featureScores = pd.concat([dfcolumns,dfscores, dfpvalues],axis=1)
...
display(feature_sorted.head(show_df))
ax = sns.barplot(x='Feature', y=col, data=feature_sorted.iloc[:show_plot],
palette="Set2")
```



Feature	Score	p-values
risk_t-0	3369.183573	0.000000e+00
risk_t-1	1956.957354	0.000000e+00
risk_t-2	1168.375352	1.090319e-237
risk_t-3	773.360845	7.070542e-162
risk_t-4	504.209945	4.872855e-108
risk_t-5	309.777849	6.068453e-68
risk_t-6	187.169852	4.337859e-42

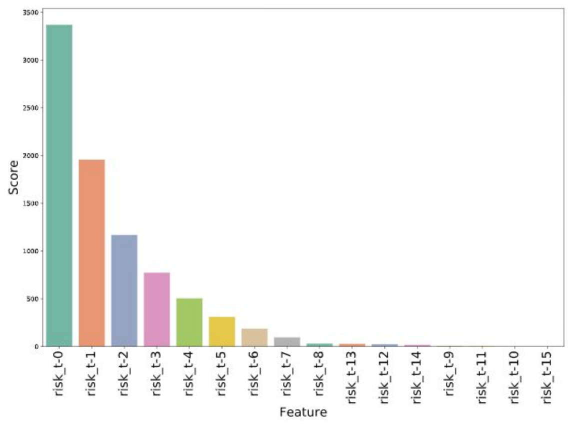


Figure 9. - Filter method: Univariate Selection

4.2. Wrapper Methods: RFE & RFE-CV

For Recursive Feature Elimination (RFE) We used the original model and other with cross-validation, both models are sklearn's function: RFE and RFECV. They require also a model to fit them. We used some of them (Logistic Regression, RandomForest, Lasso and Ridge) to compare the results and select the best features in common. Then for the cross-validation method, we plotted the score by the number of features fitted. Here is the implementation of the second because the first one is the same but shorter.

```
mse=make_scorer(mean_squared_error)
model = LinearRegression()
k_fold = 5
rfecv = RFECV(estimator=model,step=1, cv=KFold(k_fold), scoring=mse,
verbose=1)
rfecv.fit(X, y)
...
ax = sns.lineplot(x=range(1, len(rfecv.grid_scores_) + 1),
y=rfecv.grid_scores_, palette="Set2")
```

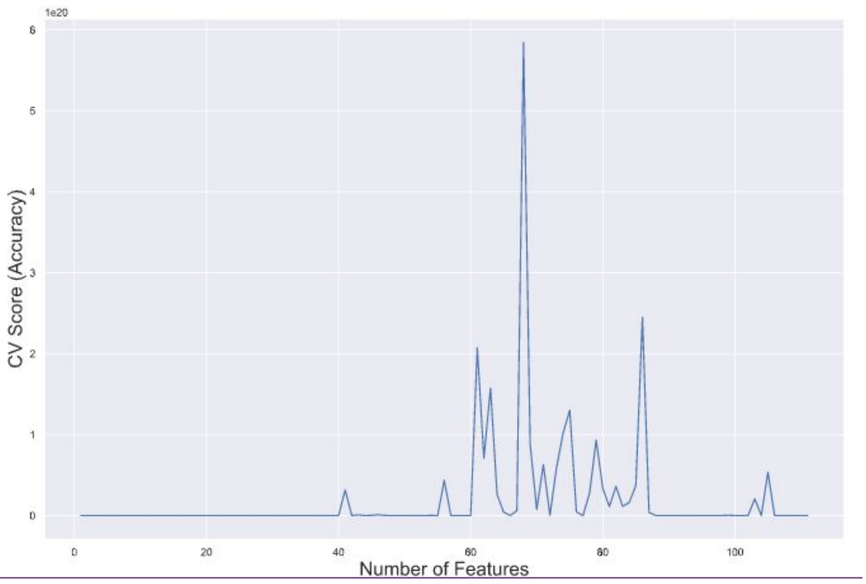
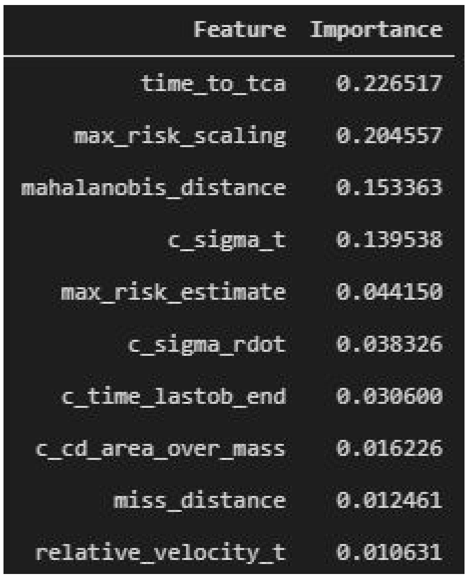


Figure 10. - Wrapper method: RFECV

4.3. Embedded Methods: Feature Importance

We applied feature importance which uses in-built class parameters from *Sklearn* models such as tree algorithms. In this case, we have used (DecisionTreeRegressor, ExtraTreesRegressor and RandomForestRegressor) to asses the importance of the features and then we stored the result and plotted it as in the univariate example.

```
model = DecisionTreeRegressor()
model.fit(X,y)

#Creating dataset of feature importances for better visualization
dfimportance = pd.DataFrame(model.feature_importances_)
...
```



Feature	Importance
time_to_tca	0.226517
max_risk_scaling	0.204557
mahalanobis_distance	0.153363
c_sigma_t	0.139538
max_risk_estimate	0.044150
c_sigma_rdot	0.038326
c_time_lastob_end	0.030600
c_cd_area_over_mass	0.016226
miss_distance	0.012461
relative_velocity_t	0.010631

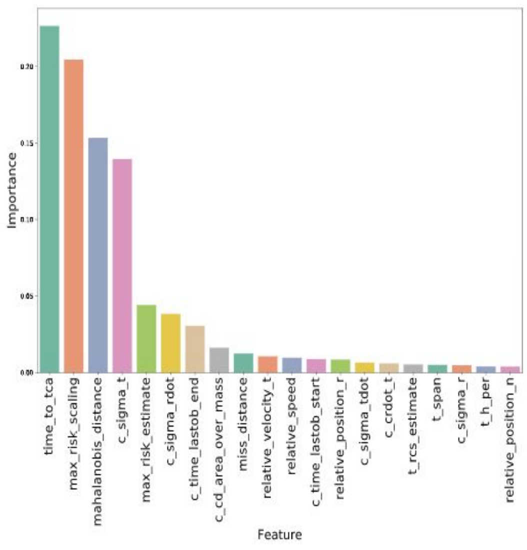


Figure 11. - Embedded method: Feature Importance

5. Feature Engineering

In this section, we discuss the implementation of feature engineering techniques. Overall, it was an easy process thanks to powerful Python libraries such as *Featuretools* and *Pandas*.

5.1. Brute Force

In this approach, we wanted to create as many features as possible in an easy way. For doing so, we used the library **Featuretools**. This library is utilized to generate myriads of features across multiple data frames by brute force. It combines the input features with aggregation and transformation operations using primitives (e.g. multiplication).

In our case, we had one dataframe so we could only use transformation operations. The primitives selected were from the list of implemented ones in the library and from some function that we created. We also set the number of total output features that we wanted as trying to generate all of them was impossible. We also set the number of CPU cores to the maximum value (it is really time-consuming). Here is a summary of the implementation:

```
# Make an entity set and add the entity
es = ft.EntitySet(id = 'events')
es.entity_from_dataframe(entity_id = 'data', dataframe = df)

# Our own transformation primitives
def Log(column):
    return np.log(column)
...

# Create the primitives
log_prim = make_trans_primitive(function=Log, input_types=[Numeric],
                                return_type=Numeric)
...

# Run deep feature synthesis with transformation primitives
df, _ = ft.dfs(entityset=es, target_entity='data',
               trans_primitives=['add_numeric', log_prim, ...],
               max_features=2000, n_jobs=-1)
```

After obtaining this large number of features, it is likely that some of them include infinite, missing, or a single constant value. To remove them, we used pandas functions. Usually, this cleaning procedure reduced the number of features by 10%. Here is the procedure:

```
# Removing columns with missing and infinite values
df = df.replace([np.inf, -np.inf], np.nan)
df.dropna(axis=1, how="any", inplace=True)
# Removing invariant columns
df = df.loc[:, (df != df.iloc[0]).any()]
```

5.2. Information on Events

As explained in the PRDL report, we created new features by extracting relevant information from the events. In particular, we extracted it from the features *risk* and *time_to_tca* as they are the most important ones.

The implementation was based on **Pandas** functions *groupby* (to group the messages or rows by event) and *transform* (to create the new feature based on an input function). As all needed functions were easy to implement, we utilized *lambda* functions. The complete implementation of these features is as follows:

```
# Counting how many instances each event has (sequence length)
df["event_length"] = df.groupby('event_id')['event_id']
                       .transform('value_counts')

# Counting how many high risk instances each event has
df["high_risk_count"] = df.groupby('event_id')['risk']
                          .transform(lambda x: (x > -6.0).sum())

# Computing mean tca jump each event has
df["mean_tca_jump"] = df.groupby('event_id')['time_to_tca']
                        .transform(lambda x: x.diff().mean())

# Computing mean risk jump each event has
df["mean_risk_jump"] = df.groupby('event_id')['risk']
                         .transform(lambda x: x.diff().mean())

# Counting positive risk jumps each event has (from high to low)
df["pos_risk_jumps"] = df.groupby('event_id')['risk']
                        .transform(lambda x: (x.diff() > 0.0).sum())

# Counting positive risk jumps each event has (from low to high)
df["neg_risk_jumps"] = df.groupby('event_id')['risk']
                        .transform(lambda x: (x.diff() < 0.0).sum())
```

Some of the new features have missing values on events that has only one row since the operation *diff* needs at least two rows to work. For those events, we padded their missing values with zeros.

```
# Fill NaNs
values = {'mean_tca_jump': 0, 'mean_risk_jump': 0}
df.fillna(value=values, inplace=True)
```

6. Deep Learning Modeling

In this section, we discuss the implementation of deep learning architectures. The selected Python library for doing so is **Keras**. It offers the easiest framework to use among all options (TensorFlow, Pytorch, MXNet, etc.) and, on top of that, it is really powerful. We used its most up-to-date implementation fully included in the TensorFlow 2.0 library.

Apart from that, we talk about some previously required processing such as data scaling, and splitting; the implementation of the evaluation metric; and the automation of hyperparameter tuning using the library *Keras-tuner*. In this section, most of the code is simplified for better interpretation.

6.1. Data Preparation

6.1.1. Data Scaling

As mention in section 2.1., data scaling is performed during the creation of the input array 'X' and the target feature 'y' to ease the padding process. To be precise, it is carried out after getting the 'y' array from the data frame (for the target feature), and after slicing the data frame to the required *time_to_tca* range.

The chosen Python library is **Scikit-learn** as it is really is to implement. We decided to have a scaler for X and another for y to ease the inverse transformation of the latter during the model evaluation. The initialization is done at the beginning of the script as shown in section 2.1. Here is a code snippet of the implementation:

```
# Scaling y using the whole risk feature after getting it from df
_ = y_scaler.fit(df["risk"].values.reshape(-1, 1))
y = y_scaler.transform(y)
...

# Scaling X as df after slicing df using X condition: time_to_tca > 2.0
df = pd.DataFrame(X_scaler.fit_transform(df), columns=df.columns)
```

6.1.2. Data Splitting & Problem Modeling

For splitting each of our X and y arrays into train, validation and test set, we used again a function of the **Scikit-learn** library called *train_test_split*, alongside some boolean operation for stratifying the process. First, we split the original arrays into a train set and test set, and then, the obtained train set into a new train set and validation set. As both operations are identical, we show just one of them in the following code snippet. Bear in mind that all variables are initialized at the beginning of the script.

```
# Splitting arrays into train and test
# Abbreviated names to fit document -> tr: train, te: test
y_boolean = (y > threshold_scaled).reshape(-1,1)
```

```
X_tr, X_te, y_tr, y_te = train_test_split(X, y,
                                          stratify=y_boolean,
                                          shuffle=True,
                                          random_state=seed,
                                          test_size = test_split)
```

After performing both operations, we have the arrays prepared for a regression problem. As explained in the PRDL report, we normally model it as classification. To do so, there is a simple extra step, in which we cast the target features “y’s” into boolean arrays using the scaled high-risk threshold since the arrays at this point are scaled too. The original threshold is scaled using the ‘y’ scaler fitted before. Here is the code implementation:

```
# Transforming y’s for classification tasks
y_train = (y_train > threshold_scaled).reshape(-1,1)
y_val   = (y_val   > threshold_scaled).reshape(-1,1)
y_test  = (y_test  > threshold_scaled).reshape(-1,1)
```

6.1.3. Hyperparameter Tuning

We automated the hyperparameter tuning process Random Search using the official Keras tuning library, *Keras-Tuner*. First, we have to create a function that returns the Keras model and has as a parameter *hp*, for the hyperparameters. Inside this function, we add some specific lines for the hyperparameters that we want to tune depending on their type (integer, float, list, etc.). It is the same for the rest of the model as in any conventional Keras model. The implementation of an LSTM model is as follows:

```
# Model function with hp parameter for the search space
def build_model(hp):

  # Searching space for the number of layers (int)
  for i in range(hp.Int('n_layers', 1, 3)):
        model.add(LSTM(...))

  # Searching space for LSTM units (int) and dropout (float)
  model.add(LSTM(units=hp.Int('units', min_value=4,
                              max_value=128, step=4),
                 dropout=hp.Float('dropout', min_value=0.,
                                  max_value=0.4,step=0.1))

  # Searching space for optimizer's learning rate (list)
  model.compile(optimizer=Adam(hp.Choice('learning_rate'
                                         [2e-3, 1e-3, 5e-4])))

  # Returning the model
  return model
```

Note that there are parts of the code that were omitted to ease the interpretability. For integers and floats such as LSTM units or dropout, we defined (i) a name to identify the hyperparameter, (ii) the minimum and maximum value of the searching space of that variable, and (iii) the step for navigating it. For lists of possible values as in the optimizer's learning rate, we simply define a name and the list.

The next part is to initialize the tuner. In our case, we use Random Search. Then, we define (i) what we want to optimize (validation accuracy, train loss, etc.), (ii) the number of configurations or trials, and (iii) the number of execution for each configuration as some configuration might behave randomly and produce a wide range of results. The code is as follows:

```
# Initializing tuner
tuner = RandomSearch(build_model, objective='val_accuracy',
                     max_trials=10, executions_per_trial=2)

# Summary of the search space
tuner.search_space_summary()
```

Finally, we fit the model. The function for doing so is called *search* but it works the same as the *fit* function of any conventional Karas model. Here, we pass the train and validation sets, the number of epochs, the batch size, class weights, etc. After fitting the data, a training summary and the best model already fitted can be accessed easily.

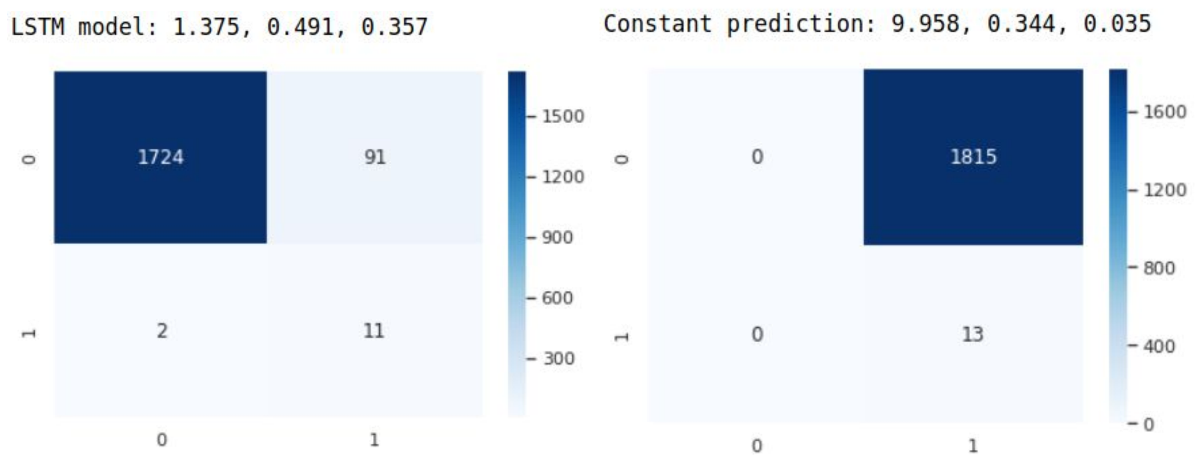
```
# Fitting data for generated search space
tuner.search(x=X_train, y=y_train,
             epochs=epochs, batch_size=batch,
             validation_data=(X_val, y_val))

# Showing training results
tuner.results_summary()

# Getting the best model and showing its architecture
best_model = tuner.get_best_models()[0]
best_model.summary()
```

6.2. Evaluation Tools

As explained in the PRDL report, the evaluation metric is unconventional. So we have to hard-code parts of it. To do so, we defined a function called *evaluate* to pass the model prediction array and the two baselines. We use mainly **Numpy** operation, **Scikit-learn** function and **Seaborn** for plotting the confusion matrix. We thought about automating the storing of all results in a dataset to compare them later on, but

Here is a simplified version of the code as the full implementation is a bit confusing. We also attach the results of running the code. Bear in mind that it is for a classification problem, so we substituted the boolean predictions for high-risk and low-risk constant values.

```python
# Evaluate function with parameters name, boolean prediction array
# and numeric test array already re-scaled
def evaluate(name, y_num_test, y_bool_pred):

    # Getting numeric predictions from boolean prediction array
    y_num_pred = y_num_test.copy()
    y_num_pred[y_bool_pred==True]  = high_risk_value
    y_num_pred[y_bool_pred==False] = low_risk_value

    # Computing mse with high-risk events in test set only
    y_mse_pred = y_num_pred[np.where(y_num_test >= -6.0)]
    y_mse_test = y_num_test[np.where(y_num_test >= -6.0)]
    mse = mean_squared_error(y_mse_test, y_mse_pred)

    # Computing f-beta with all boolean prediction
    y_bool_test = (y_num_test >= -6.0).reshape(-1,1)
    f_beta = fbeta_score(y_bool_test, y_bool_pred, 2)

    # Computing the evaluatoin metric
    score = mse / f_beta

    # Plotting results
    print(name, "{:0.3f}, {:0.3f}, {:0.3f}".format(score, mse, f_beta))
    sns.heatmap(confusion_matrix(y_bool_test, y_bool_pred))
    plt.show()
```



```
LSTM model: 1.375, 0.491, 0.357
Constant prediction: 9.958, 0.344, 0.035
```

Figure 12. - Model evaluation results for an LSTM model and the competition baseline.

6.3. Deep Learning Models

6.3.1. FeedForward Neural Networks

As explained in the PRDL report, for FFN-based models only the last CDM (row) of each event (with *time_to_tca* greater than 2 days) was used as input. To define the architecture and other hyperparameters, we have used simply trial and error, due to lack of time to implement more complex optimization methods. Finally the neural structure definition have been implemented in the following way:

```
# Model
input_shape = X_train_scaled.shape[1]
inputs_1 = Input(shape=(input_shape))
x = Dense(256, activation='relu',
          kernel_regularizer=regularizers.l2(0.001))(inputs_1)
x = Dense(128, activation='relu',
          kernel_regularizer=regularizers.l2(0.001))(x)
x = Dense(64, activation='relu',
          kernel_regularizer=regularizers.l2(0.001))(x)
x = Dense(16, activation='relu',
          kernel_regularizer=regularizers.l2(0.001))(x)
x = Dense(4, activation='relu',
          kernel_regularizer=regularizers.l2(0.001))(x)
output_1 = Dense(1, activation = 'sigmoid')(x)

model_ffn = Model(inputs=inputs_1, outputs = output_1)
opt = tfa.optimizers.RectifiedAdam()
loss_ = losses.binary_crossentropy

model_ffn.compile(optimizer=opt,loss = loss_, metrics =['accuracy'])
```

The training is then carried out with the *fit* function of the *Keras* models. For obtaining satisfactory results it was necessary to include a weight to the different classes of the dataset during this training. This is indicated in the parameter *class_weights*, where a dictionary with the weights has been defined.

```
class_weights = {0: 2,
                 1: 90}
batch_size = 32
epochs = 60
history = model_ffn.fit(X_train_scaled, y_train, shuffle=True,
                        validation_data=(X_val_scaled, y_val),
                        batch_size=batch_size, epochs=epochs,
                        class_weight=class_weights)
```

For the rest of the neuronal models this function, which executes the training with its specific parameters, is implemented in a very similar way so it will not be included in more occasions.

6.3.2. Recurrent Neural Networks

From now on, we will focus only on the architecture of each model since the rest of the steps are shared, with some exceptions. As for the Recurrent Neural Networks, the architecture is very similar to the previous one. Here is an example of a small and constrained network using the Keras' functional API.

```
# Defining LSTM model
input_tensor = Input(batch_shape=(batch, timestep, X_train.shape[2]))

rnn_1 = LSTM(32, stateful=False, recurrent_dropout=0.3,
             dropout=0.15, return_sequences=True,
             kernel_regularizer=L1L2(l1=0.0, l2=0.01))(input_tensor)

batch_1 = BatchNormalization()(rnn_1)

output_tensor = Dense(units = 1, activation='sigmoid')(batch_1)

model = Model(inputs=input_tensor, outputs= output_tensor)
```

6.3.3. Convolutional Neural Networks

The convolutional neural networks have also a pretty straightforward implementation. As stated in the PRDL report, we try to resemble VGG-16 blocks but using one-dimensional filters. Here is a small model that uses just one of these blocks using the sequential API this time.

```
# Defining Conv1D model
model = Sequential()
model.add(Input(batch_shape=(batch, timestep, X_train.shape[2])))

# VGG16-like block
model.add(Conv1D(10, 3, activation='relu'))
model.add(Conv1D(10, 3, activation='relu'))
model.add(MaxPooling1D(2))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

6.3.4. Autoencoders

For implementing our LSTM Autoeconders, we needed a few extra steps for generating the train and test set (the validation set is passed as a percentage to Keras fit function) and for evaluating the model. First, for generating the sets, we split the data frame into train and test before getting the 3D-dimensional array. This time, there is no 'y' target feature as now the input array is also the target array that we want to reconstruct. Here is the implementation to split the data frame:

```
# Creating feature with last value of sequence for each event (former y)
df["last_value"] = df.groupby(["event_id"])["risk"]
                     .transform(lambda x: x.iloc[-1])

#Splitting dataframe into train and test
df_train = df.loc[df["last_value"]<high_risk_threshold]
df_test = df.loc[df["last_value"]>=high_risk_threshold]
```

The resulting dataframes are processed as in previous cases to get the three-dimensional arrays. The implementation of the Keras model is again really straightforward. It is similar to RNNs but needs some specific layers for the 'bottleneck' (the layer between encoder and decoder) and for the output. Here is an example using the sequential API:

```
# Defining LSTM autoencoder model
lstm_autoencoder = Sequential()

# Encoder
model.add(LSTM(32, activation='selu', stateful=False,
               input_shape=(timestep, X_train.shape[2]),
               return_sequences=True))
model.add(LSTM(16, dropout=0.1, stateful=False,
               return_sequences=False))
model.add(RepeatVector(timestep))

# Decoder
model.add(LSTM(16, dropout=0.1, stateful=False,
               return_sequences=True))
model.add(LSTM(32, activation='selu', stateful=False,
               return_sequences=True))
model.add(TimeDistributed(Dense(X_train.shape[2])))
```

Finally, the evaluation of an autoencoder for anomaly detection is based on plotting the distribution of the reconstructed loss error (in our case MSE) of the train (low-risk events) and test (high-risk events) and putting a threshold to categorized the classes. Normally, regression models return a one-dimensional array that is used to compute the loss function with the test set. This time, we do not have a one-dimensional array but a three-dimensional array (the reconstructed input) that has many timesteps and features. We decided to compute the mse only using the last timestamp of each event's risk feature, as it has been always the most important timestamp and feature. We thought about a more profound analysis selecting several timestamps and features but ultimately discarded it due to its high complexity. Here is the simplified implementation of the model evaluation:

```
def evaluate(X_real):

    # Getting prediction
    X_pred = lstm_autoencoder.predict(X_real, batch_size=batch)

    # Computing the mse with just the last timestamp of risk feature
    X_real = X_real[:, timesteps, risk_feature_index]
    X_pred = X_pred[:, timesteps, risk_feature_index]
    mse = np.mean(np.power(X_real-X_pred, 2), axis=1)

    # Plotting the reconstructed mse distribution
    sns.distplot(mse)
    plt.show()

    return mse

reconstructed_mse_train = evaluate(X_train)
reconstructed_mse_test  = evaluate(X_test)
```

6.3.5. Siamese Neural Networks

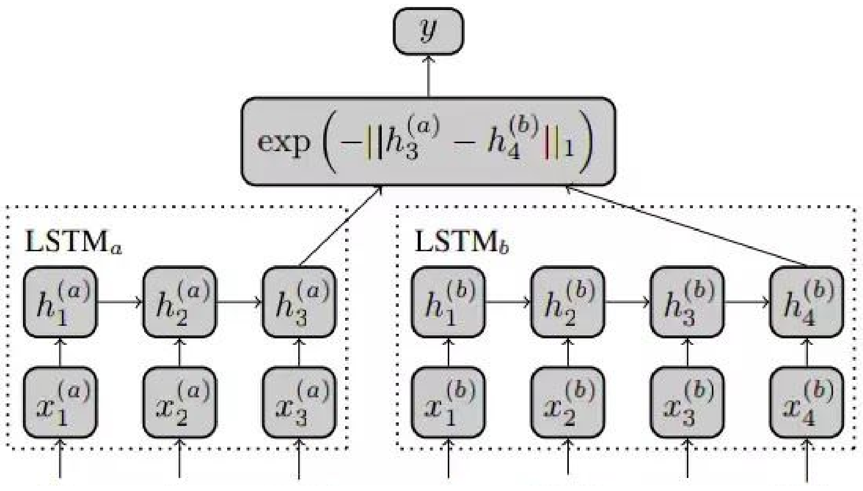
Siamese neural networks were the most difficult to implement. They required advanced Numpy indexing and meticulous steps to create as many pairs as possible and for evaluating the model by comparing each event in the test with the whole train or validation sets. Moreover, the code is nothing but short.

For these reasons, we have decided to not explain in this report the details of these steps and just focus on the deep learning architecture. If the readers still want to see the implementation, please visit our Github **[1]** account and read the reasoning behind in the PRDL report.

As for the architecture, we used a couple of articles to implement it **[4][5]**. The first part is to define the Manhattan distance. We use the Keras backend operation (sum, exp, abs, etc.) for computing the distance. Then, we define both inputs left and right with the functional API. These inputs are used to feed the same LSTM model that is defined using a sequential object but then is computed as a part of the functional-based model.

After feeding both inputs, we get out two outputs. To compute the manhattan distance previously defined for these outputs, Keras allows the creation of new layers by using a Lambda layer in which you pass the implemented function (the manhattan distance). Finally, we create the model object as in any other functional case. Here is the full code of the model and a figure of what has been implemented:

```
# Defining Manhattan distance
def manhattan_distance(left, right):
    return K.exp(-K.sum(K.abs(left-right), axis=1, keepdims=True))

# Defining Manhattan LSTM model
left_input = Input((timestep, features))
right_input = Input((timestep, features))

lstm = Sequential([LSTM(32, stateful=False, dropout=0.25,
                        recurrent_dropout=0.5, return_sequences=True,
                        input_shape=(timestep, features)),
                   BatchNormalization(),
                   LSTM(16, stateful=False, dropout=0.25,
                        recurrent_dropout=0.5, return_sequences=False),
                 ])

left_output = shared_lstm(left_input)
right_output = shared_lstm(right_input)

distance = Lambda(function=lambda x: manhattan_distance(x[0], x[1]),
                  output_shape=lambda x: (x[0][0], 1)
                 )([left_output, right_output])

model = Model([left_input, right_input], [distance])
```



y
$\exp\left(-\|h_3^{(a)} - h_4^{(b)}\|_1\right)$
LSTM_a
LSTM_b
$h_1^{(a)}$ $h_2^{(a)}$ $h_3^{(a)}$
$x_1^{(a)}$ $x_2^{(a)}$ $x_3^{(a)}$
$h_1^{(b)}$ $h_2^{(b)}$ $h_3^{(b)}$ $h_4^{(b)}$
$x_1^{(b)}$ $x_2^{(b)}$ $x_3^{(b)}$ $x_4^{(b)}$

Figure 13 - Manhattan LSTM architecture example.

6.3.6. Deep Ensembles

As explained in the PRDL report, in order to improve the results and especially the robustness of the models, it was decided to implement ensemble methods. For the implementation, the Python library *DeepStack* was used, which made the process much easier. Although it is noted that it is a very new library with few functionalities yet. In our case, we have used 5 pre-trained FFN models as input, and a stacking meta-model based on Random Forest. First, we must import the trained neural models with the Keras' function *load_model*, which we have previously saved in ".h5" format.

```
model_1 = load_model('model_1.h5')
model_2 = load_model('model_2.h5')
...
```

Then, we import these models as KerasMembers. The most interesting part of this library is probably in this step. As we can see, we can indicate the training and validation sets for each model. This is because it is not necessary that the models have been trained with exactly the same dataset, providing a lot of flexibility. Although it is necessary that the models have the same class labels (*y_train* and *y_val*)

```
member_1 = KerasMember(name="model_1", keras_model=model_1,
                       train_batches=(X_train, y_train),
                       val_batches=(X_val, y_val))
member_2 = KerasMember(name="model_2", keras_model=model_2,
                       train_batches=(X_train, y_train),
                       val_batches=(X_val, y_val))
...
```

Now we create the stack model and train it with the *fit* function.

```
from deepstack.ensemble import StackEnsemble
class_weights = {0: 300,
                 1: 1}
stack = StackEnsemble()
stack.model = RandomForestClassifier(verbose=0,
                       n_estimators=200,class_weight=class_weights)
stack.add_members([member1, member2, ...])
stack.fit()
stack.describe()
```

Finally we predict with the obtained model, introducing as input *X_test_stack*, which is the union of the prediction arrays obtained by the neuronal models used as input.

```
stack_deep = stack.predict(X = X_test_stack)
```

7. Conclusions

In this project, we have faced a real challenge implementation-wise. Due to the rawness of the dataset, there were many steps that needed a lot of thinking and understanding, for the design and implementation of the Deep Learning models. We truly think that our previous experience working with this technology has played a big role in this project. It was not by any means for inexperienced programmers.

Aside from the difficulties of processing the dataset, the implementation of deep learning models was really straightforward as we were already familiar with the technology and libraries. Keras is an incredible source to work due to its powerful high-level and user-friendly implementation. Most of the time while creating our models, we could focus on their design instead of their implementation as we found it really easy.

We also want to highlight two new Python libraries that have great potential, *Featuretools* and *Keras-tuner*. The former allows the creation of myriads of features in the span of a second. The former allows the implementation of an easy way to perform hyperparameter tuning. We have only utilized the Random Search method, but it also includes other impressive options such as Bayesian optimization that we hope to use in the future.

In brief, the most challenging parts were the creation of the dataset 'versions', and the extra processing and model evaluation of both autoencoders and siamese networks. To the best of our knowledge, we tried to design, implement and document everything in the easiest, and most efficient way possible, in an attempt to write code that could be used in future projects.

To conclude this report, we want to remind the reader that the full implementation of the code will be published shortly in our Github accounts **[1][2][3]**.

8. References

[1] Github, "Sergio Pérez Morillo Profile," [Online]. Available:
https://github.com/spmorillo
[Accessed 10 January 2020]

[2] Github, "Jaime Pérez Sánchez Profile," [Online]. Available:
https://github.com/jaimeperezsanchez
[Accessed 10 January 2020]

[3] Github, "Carlos Andrés Ramiro Profile," [Online]. Available:
https://github.com/carlosaramiro
[Accessed 10 January 2020]

[4] Medium, "How to predict Quora Question Pairs using Siamese Manhattan LSTM," [Online]. Available:
https://medium.com/mlreview/implementing-malstm-on-kaggles-quora-question-pairs-competition-8b31b0b16a07
[Accessed 10 January 2020]

[5] Towards Data Science, "Predictive Maintenance with LSTM Siamese Network," [Online]. Available:
https://towardsdatascience.com/predictive-maintenance-with-lstm-siamese-network-51ee7df29767
[Accessed 10 January 2020]