ID2214 Assignment 4:
Activity Prediction for Chemical Compounds

Roeland Hooijmans
Mattia Evangelisti
Gustav Normelli
roelandh | mattiaev | normelli @kth.se

15 December 2022

Contents

1 Introduction

In this assignment, we are given a dataset with chemical compounds represented by strings of text in Simplified Molecular Input Line Entry Specification (SMILES) notation. This set of 156,258 chemical compounds has been used to develop a predictive model trained on recognizing if a compound is biologically active or inactive. From the SMILES representations, objects of molecules were added to the dataset using a function from the `RDKit`[5] library. From these objects, further extraction of features using the `RDKit` was performed to get a full dataset. As a curiosity, it can be mentioned that the `RDKit` can be used for making drawings of molecules like the one in Figure 1.

Figure 1: Example of molecule object

The approach for the development of the predictive model was, to begin with getting to know the data at hand. To do so we started with an exploratory data analysis (EDA) to get an initial understanding of the main characteristics of the data. This was simultaneously done with feature selection and data preparation before choosing which standard models to use. Next, after picking the model it was trained and later tuned by exploring different tuning techniques. The flow of this process can be viewed in Figure 2 below. All this was done using the Python libraries from Numpy [1], Pandas [8], scikit-learn [4], imblearn [3], scikit-optimize [2].

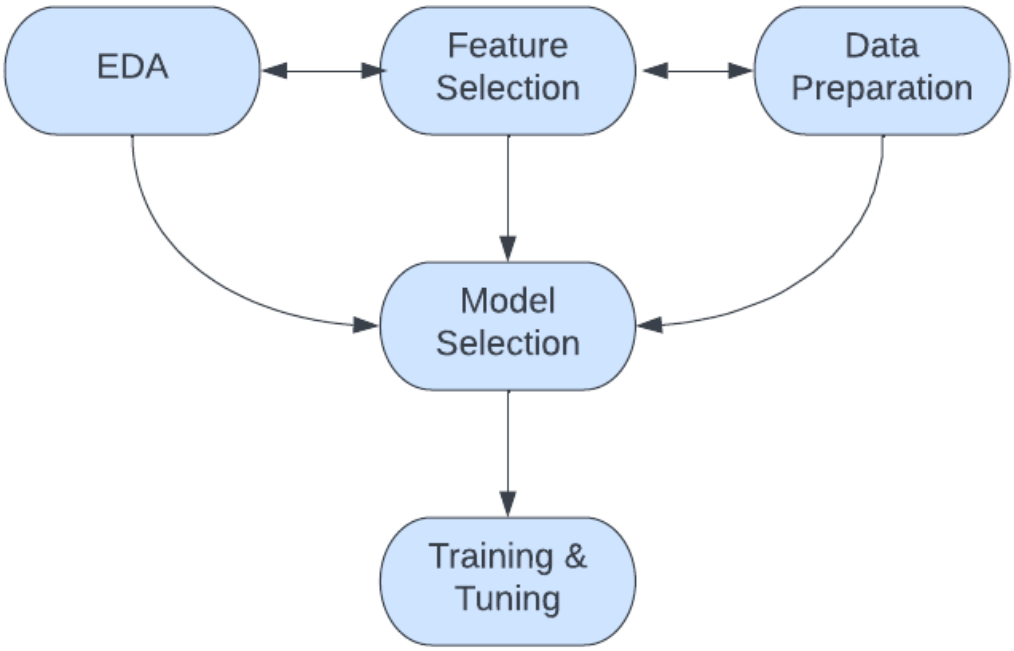

Figure 2: Process flow for the development of the predictive model

2 Feature selection & Data preparation

This section includes a description of how the feature selection and the data preparation have been done before finding the best-performing model. It includes a description of what techniques for feature selection were used and how the problem with an imbalanced dataset was solved.

2.1 Feature Selection

Feature selection is one of the most crucial steps in a classification problem. To select the most relevant features, we first extracted a large number of features (about 30) using the `RDKit` [5] library functions. The choices of the initial features were made after the library documentation has been read. All the selected features could be added directly to a data frame apart from one, the Morgan Fingerprint, which was represented as a bit vector and had to be transformed into a string object to be used properly. Then two feature selection techniques were applied:

- `Univariate selection`: a technique for examining each feature individually to determine the strength of the relationship of the features with the response

variable. In our case, SelectKBest class, from `scikit-learn` [4] library was used. This class can be used with a suite of different statistical tests to select a specific number of features, we chose the chi-squared test for non-negative features. This test measures dependence between variables, to eliminate the features that are the most likely to be independent of class and therefore irrelevant to classification. In Table 1 below is the selection of the 10 best features.

`morgan_fingerprint`	$4.239752e^{124}$
`exact_mol_wt`	13266.8
`num_valence_electrons`	4003
`num_bonds`	1271.68
`num_heavy_atoms`	945.522
`num_atoms`	945.482
`num_aromatic_rings`	277.587
`benzene`	128.534
`amide`	102.746
`NHOH_count`	67.6791

Table 1: Top 10 feature selected

- `Feature importance`: the importance of each feature of a datset can be obtained by using the feature importance property of the model. Feature importance gives a score for each feature of the data, the higher the score the more important or relevant the feature is to the output variable. Feature importance is an inbuilt class that comes with Tree-Based Classifiers. We chose to use Extra Tree Classifier for extracting the top 10 features for the dataset, seen in Figure 3 below.

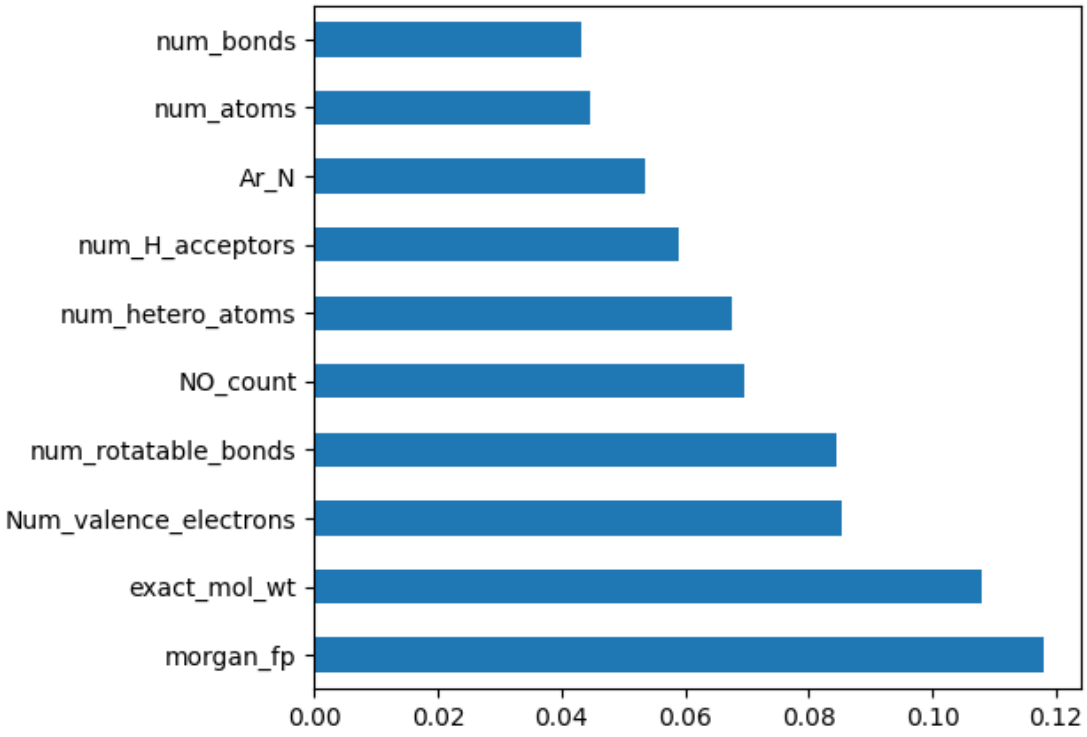

Figure 3: Top 10 features selected

After retrieving the 10 best features from each technique, we merged the two obtained sets and achieve a final result of 15 best features. We then computed the correlation matrix of this set of features. We noticed that some of them were highly correlated (e.g. `num_atoms` and `num_heavy_atoms` have a correlation equal to 1). Features with high correlation are more linearly dependent and hence have almost the same effect on the dependent variable. So, when we found pair of features highly correlated we dropped one of the two. In this case, the choice of the feature to be dropped from the pair was chosen with respect to its importance in the two techniques described above. After this analysis, we ended up with ten features, which can be viewed in the correlation matrix of Figure 4 below.

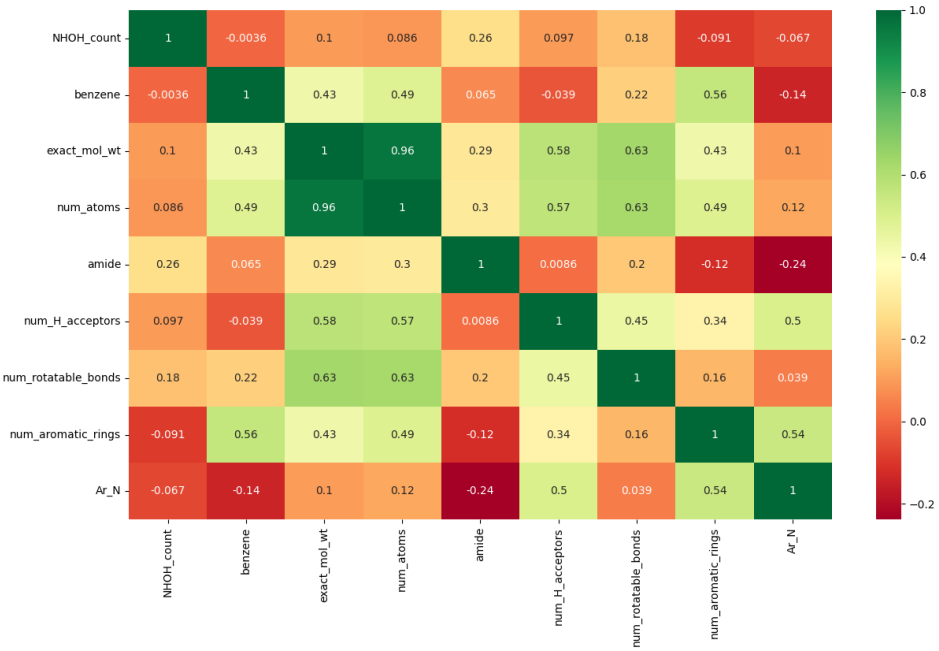

Figure 4: Correlation matrix of the selected features

As can observe from Figure 4 above, all the features present a medium or low correlation value, except for the pair `num_atoms` - `exact_mol_wt`. We chose to keep this pair given the high importance of the two features founded in the previous analysis.

In the end, we obtained the best set of features for our classification model. In particular, the Morgan Fingerprint was shown to be a good predictor of biological activity. This is also supported by a study that has shown that Morgan Fingerprints are a great predictor of biological activity [6].

Another technique applied to select the most important feature is the Recursive Feature Elimination with Cross-Validation (RFECV), which was used with a built-in function of the `scikit-learn` library. This method utilizes a machine learning model to select the features by eliminating the least important feature after recursive training. In every iteration, all the least important features are pruned from the current set of features to arrive at the most important ones.

After we applied this technique, we ended up with a set of 21 features. We did not measure any improvement in performance by applying the RFECV technique. This lead us to choose to go with the method described above, so as to have a smaller set of features and improve the computational speed of our algorithm.

2.2 Data Preparation

The data cleaning phase for this dataset was really fast since all the features were extracted using `RDKit` functions and therefore did not present any strange behavior such as missing values, null values, or outliers. The only needed action was to convert and apply the one-hot encoding to the morgan fingerprint, in order to store it as 124 different columns each having either value 1 or 0.

We defined a function called `transform`, which applies imputation, encoding, and principal component analysis (PCA) to our data. All these operations are performed using functions from the `scikit-learn` library and are performed sequentially thanks to the use of a pipeline object, which is part of the same library named before. Below is the code used to implement all these data preparation techniques:

```
def transform(Xy_train):
X_train, y_train = Xy_train

num_features = X_train.select_dtypes(include=['float64']).columns
cat_features = X_train.select_dtypes(include=
                                     ['object', 'bool']).columns
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant',
                              fill_value="missing")),
    ('encoder', OrdinalEncoder(handle_unknown='use_encoded_value',
                               unknown_value=-1))])
numeric_transformer = Pipeline(steps=[
                    ('imputer', SimpleImputer(strategy='median')),
                    ('pca', PCA(n_components=0.95))])
preprocessor = ColumnTransformer(transformers=[
                ('num', numeric_transformer, num_features),
                ('cat', categorical_transformer, cat_features)])

return preprocessor
```

This function applies all the necessary pre-processing to the data and returns a `preprocessor` object which is then used before applying any model. The normalization is, on the other hand, applied before the model, using again a pipeline object:

```
extreme = Pipeline(steps=[('preprocessor', preprocessor),
                          ('scaler', StandardScaler()),
                          ('extreme', ExtraTreesClassifier())])
```

The pipeline allows us to define a sequence of operations and its order, by doing that we can apply that imputation, encoding, PCA, and normalization to the data before the model is trained or tested.

After an intensive data analysis, we notice that the dataset is extremely imbalanced, as can be seen in the image below.

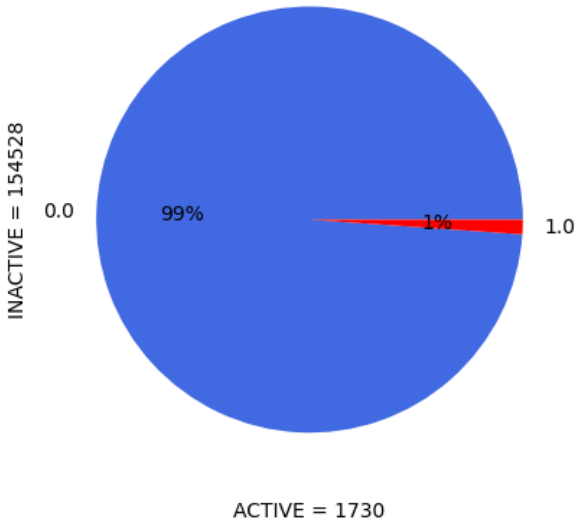

Figure 5: Label distributions

Based on this result, we could choose to apply five different types of data preparation techniques suitable to deal with the imbalanced datasets:

- **Do nothing**: one of the most common ways to deal with an imbalanced data set is to leave it as it is since very often the problem is not in the dataset itself but in the choice of the model and the metric.
- **Undersampling**: with this technique, we undersample the majority class, i.e., we reduce the number of instances of the majority class (the non-active class in our case) until we reach the point when both classes have the same number of instances equal to the one of the minority class.
- **Oversampling**: with this technique, we oversample the minority class, i.e., we increment the number of instances of the minority class (the active class in our case) until we reach the point when both classes have the same number of instances equal to the one of the majority class.
- **Synthetic samples**: In this technique, most known as SMOTE (Synthetic Minority Oversampling Technique) the minority class is oversampled by creating "synthetic" instances rather than by oversampling with replacement. In other words, with this technique, we are creating "new" instances starting from the original ones in the minority class.

We chose to go with the SMOTE technique becuase we noticed that oversampling was the better solution in terms of the AUC score and F1 score. Instead of randomly upsampling and creating copies of existing samples, it creates new synthetic samples starting from the ones already existing in the minority class. SMOTE is available in the library `imblearn` [3]. The code implementing this technique will be shown and explained in the following sections.

3 Classification

3.1 Model Selection

At this point, it is time to select an appropriate model for this classification problem. We selected and tested a set of different models as a first step in the attempt to figure out which performed the best.

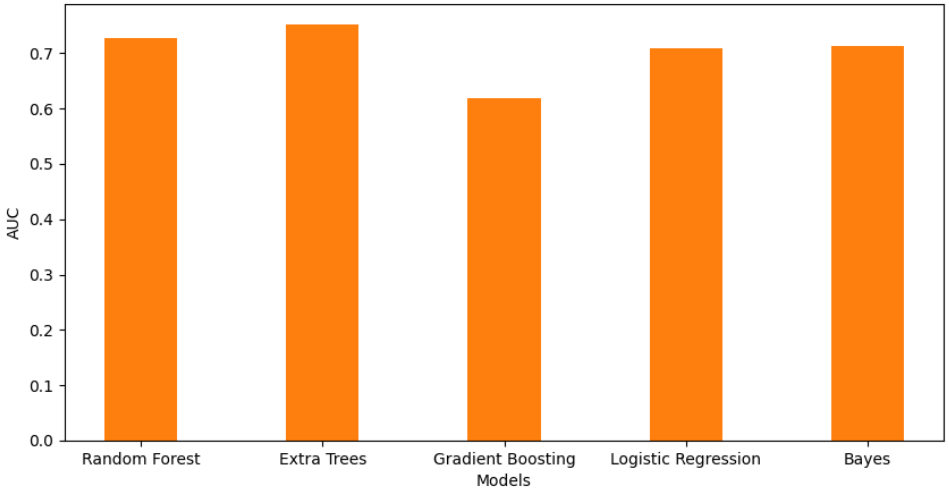

Figure 6: Mean AUC of tested models

All the models were tested without tuning and the best performing was the `ExtraTreesClassifier`. The results are displayed in Figure 6. The class `ExtraTreesClassifierfrom` the `scikit-learn` library, is an estimator that fits several randomized decision trees on various bootstrap replicas of the dataset and user averaging to improve the predictive accuracy and control over-fitting. Thanks to the high level of randomness and the binary tree nature, this model is perfect for classification over an imbalanced dataset.

3.2 Training & Tuning

For the training and tuning phases, we decided not to manually split the original dataset into train and test subsets, but rather to use the k-fold stratified cross-validation. This technique, typically used for imbalanced datasets, allows us to split the dataset into train and test sets preserving the class ratio of our target variable. This was implemented using the `StratifiedKFold` package available in the `scikit-learn` library.

```
cv = StratifiedKFold(n_splits=5, shuffle=True)
```

We use $k = 5$, which will produce 5 folds and we set shuffle to true to have even more reliable results. By doing that, at each iteration, the model is trained on one 4 folds and tested on the remaining one and then the average accuracy of each iteration is considered.

As explained in the previous sections, we decided to use the SMOTE oversampling technique and the `ExtraTreesClassifier`, now we need to tune the hyperparameters of this model. We explored three different techniques for tuning: `GridSearchCV`, `RandomSearchCV`, `BayesSearchCV`, the first two are from the `scikit-learn` library, while the last one is available in the `scikit-optimize` package. All three methods explore a set of parameters and return the best combination for the required score. The grid search explores all possible combinations of the given parameters, the random search randomly chooses a set of parameters every iteration, and the Bayesian search optimizes its parameter selection in each round according to the previous round score. Below is the code used for the creation and tuning of the model:

```
def upsampling(preprocessor, Xy):
    X, y = Xy
    cv = StratifiedKFold(n_splits=5, shuffle=True)

    pipeline = make_pipeline(preprocessor,
                             StandardScaler(),
                             SMOTE(random_state=42),
                             ExtraTreesClassifier(random_state=42))

    cross_val_score(pipeline, X, y, scoring = 'roc_auc', cv=cv)
    params = {...}

    search = RandomizedSearchCV(pipeline, param_distribution=params,
                                cv=cv, scoring='roc_auc', n_iter=20)
    search.fit(X, y)
    search.best_params_
    search.best_score_
```

This code snippet explains how the tuning of the hyperparameters is carried out. The variable `params` is a dictionary containing a set of values for each parameter we tuned, it has been omitted from the code for the sake of brevity. The pipeline allowed us to perform all the necessary preprocessing and the upsampling of the dataset. Then the model is created and trained, the last two lines allow us to extract the best score and the best parameters after the tuning. Please notice that the pipeline used in this code snippet is from `imblearn` library.

After the first round of tuning using random search with a relatively small set of parameters, we obtained the following best scores in Table 2 below.

AUC	F1
83.78%	4,72%

Table 2: Results of `ExtraTreesClassifier` hyperparameters tuning

We measure the AUC, but also the F1 score, which combines precision and recall into a single metric. This metric is especially valuable when working on classification models in which the dataset is imbalanced, i.e., our case.

These results are already good, but we decided to perform some fine-tuning using the `BayesSearchCV`. Since this method chooses only the relevant search space and discards the ranges that will most likely not deliver the best solution. This result in a minimization of the tuning time, thus a wider range of values for the hyperparameters may be used. After applying what is described above, we obtain the best score in Table 3 below.

AUC	F1
84.24%	7.8%

Table 3: Results of hyperparameters tuning using `BayesSearchCV`

4 Results

After we concluded the fine-tuning and we found the best parameters, we tested the model 20 times applying each time a 5-fold cross-validation so that at each iteration the model is tested on 5 different folds, which are then averaged. In the end, we obtained an average AUC of 83.89%, while the distribution of the values can be seen in the graphs of Figure 7 below.

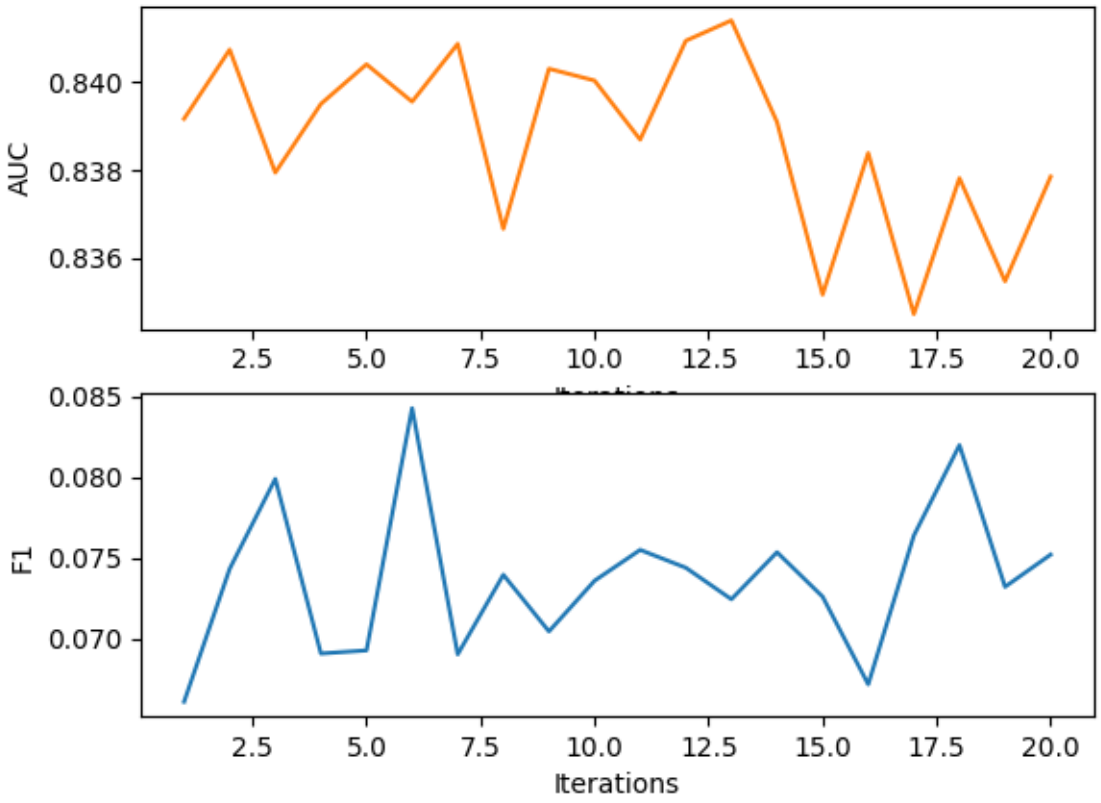

Figure 7: AUC and F1 scores over 20 interations

The last step is to fit the model with the data from the SMOTE oversampling and use the developed model to predict the estimated AUC and the class probability of the unlabelled data. Note that we have to apply the same feature selection and data preparation we used on the training data to the unlabelled dataset as well.

5 Method

In this section, the reasoning behind our choices will be summed up, to give a clear idea of what was our method while developing this project.

For the feature selection, we decided to apply two different techniques, one based on the statistical relevance of each feature, while the other considered the feature importance property of a model. We chose to do so to obtain a more solid result. Then we dropped the features which were highly correlated since features with high correlation are more linearly dependent and hence have almost the same effect on

the target variable, This whole process allowed us to drop not relevant features and lighten the dataset.

During the data preparation phase, we mainly focused on dealing with the high-class imbalance of the dataset. We explored both undersampling and undersampling and we decided to go for the latter. Even though the project would have been faster with the undersampling, since we would have reduced the size of the dataset, we decided to use upsampling since it lead to better performances of the model.

Finally, we decided on the `ExtraTreesClassifier` both because it was the best performing and because tree-based ensemble models perform better on this kind of dataset [7].

6 Discussion

The initial set of features was chosen quite arbitrarily. Since we are no experts in cheminformatics and lack domain knowledge, we decided to choose a set of features that made the most sense to us to perform the feature selection techniques on. This fact might have skewed the selection of features in one way or another. However, we are completely unaware of what features would have been theoretically correct to choose. The only requisite that was looked for was if the function to derive the feature returned an integer of float. So, there is some randomness built into it. A better way would probably have been to cast a broad net for extracting features capturing all appropriate features and the run the features selection techniques on all the features. Also, the fact that the dataset was extremely unbalanced was an issue that needed a solution. There was a relatively small amount of biologically active molecules present. Therefore, we had to manipulate our dataset. The techniques to do so can be discussed but the decisions that lead us forward were based on the results we got from the use of those techniques. The SMOTE technique was the final choice but could have been different if the goal was not to maximize AUC score and F1 score. The synthetic sample that is derived from SMOTE does not rely on randomness which could be argued to reduce bias in the dataset.

References

[1] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] Tim Head et al. *scikit-optimize/scikit-optimize: v0.5.2*. Version v0.5.2. Mar. 2018. DOI: 10.5281/zenodo.1207017. URL: https://doi.org/10.5281/zenodo.1207017.

[3] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: http://jmlr.org/papers/v18/16-365.html.

[4] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[5] *RDKit: Open-source cheminformatics*. URL: http://www.rdkit.org.

[6] Sereina Riniker et al. "Using information from historical high-throughput screens to predict active compounds". In: *Journal of Chemical Information and Modeling* 54.7 (July 2014), pp. 1880–1891. ISSN: 1549960X. DOI: 10.1021/CI500190P/SUPPL_FILE/CI500190P_SI_002.ZIP. URL: https://pubs.acs.org/doi/full/10.1021/ci500190p.

[7] Peibei Shi and Zhong Wang. "An Ensemble Tree Classifier for Highly Imbalanced Data Classification". In: *Journal of Systems Science and Complexity* 34.6 (Dec. 1, 2021), pp. 2250–2266. ISSN: 1559-7067. DOI: 10.1007/s11424-021-1038-8. URL: https://doi.org/10.1007/s11424-021-1038-8.

[8] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.