Assignment 4a Group no. 10

Project members:

Veronika Cucorova cucorova@kth.se
Tim Roelofs tjtro@kth.se
Léo Vuylsteker leov@kth.se

Assignment

In this assignment, we were tasked to develop a predictive model using a suitable representation (feature set) and learning algorithm, to predict the biological activity of chemical compounds represented by SMILES.
For this, we were given training set with 19,125 chemical compounds and their activity labels (1 or 0).

Generating representations

There are many ways to generate features from SMILES, such as through fingerprints and other descriptors. Our approach for the feature selection was to attempt to generate many potential features using the RDKit toolkit and then use a feature selection algorithm to select the best features from these.

Generating features

As stated, we used the RDKit toolkit to generate features from the data, specifically the RDKit.Chem package. We used the following sub-packages:

Module	***Description***
rdkit.Chem.Crippen	Atom-based calculation of LogP and MR using Crippen's approach
rdkit.Chem.Descriptors	Calculation of more molecular descriptors
rdkit.Chem.Fragments	functions to match a bunch of fragment descriptors
rdkit.Chem.GraphDescriptors	Calculation of topological/topochemical descriptors
rdkit.Chem.Lipinski	Calculation of Lipinski parameters for molecules
rdkit.Chem.QED	Calculation of QED (quantitative estimation of drug-likeness) parameters
rdkit.Chem.rdMolDescriptors	Calculation of molecular descriptors

From these, we used the following functions:

Function with inputs (x is the DataFrame)	***Datatype of output***	***N. of features***
rdkit.Chem.Crippen		
MolLogP(x)	float	1
MolMR(x)	float	1
rdkit.Chem.Descriptors		
ExactMolWt(x)	float	1
FpDensityMorgan1(x)	float	1
FpDensityMorgan2(x)	float	1
FpDensityMorgan3(x)	float	1
HeavyAtomMolWt(x)	float	1
MaxAbsPartialCharge(x)	float	1
MaxPartialCharge(x)	float	1
MinAbsPartialCharge(x)	float	1
MinPartialCharge(x)	float	1
MolWt(x)	float	1
NumRadicalElectrons(x)	int	1
NumValenceElectrons(x)	int	1
rdkit.Chem.Fragments		
fr_Al_COO(x)	int	1
fr_Al_OH(x)	int	1
fr_Al_OH_noTert(x)	int	1
fr_ArN(x)	int	1
fr_Ar_COO(x)	int	1
fr_Ar_N(x)	int	1
fr_Ar_NH(x)	int	1
fr_Ar_OH(x)	int	1

fr_COO(x)	int	1
fr_COO2(x)	int	1
fr_C_O(x)	int	1
fr_C_O_noCOO(x)	int	1
fr_C_S(x)	int	1
fr_HOCCN(x)	int	1
fr_Imine(x)	int	1
fr_NH0(x)	int	1
fr_NH1(x)	int	1
fr_NH2(x)	int	1
fr_N_O(x)	int	1
fr_Ndealkylation1(x)	int	1
fr_Ndealkylation2(x)	int	1
fr_Nhpyrrole(x)	int	1
fr_SH(x)	int	1
fr_aldehyde(x)	int	1
fr_alkyl_carbamate(x)	int	1
fr_alkyl_halide(x)	int	1
fr_allylic_oxid(x)	int	1
fr_amide(x)	int	1
fr_amidine(x)	int	1
fr_aniline(x)	int	1
fr_aryl_methyl(x)	int	1
fr_azide(x)	int	1
fr_azo(x)	int	1
fr_barbitur(x)	int	1
fr_benzene(x)	int	1

fr_benzodiazepine(x)	int	1
fr_bicyclic(x)	int	1
fr_diazo(x)	int	1
fr_dihydropyridine(x)	int	1
fr_epoxide(x)	int	1
fr_ester(x)	int	1
fr_ether(x)	int	1
fr_furan(x)	int	1
fr_guanido(x)	int	1
fr_halogen(x)	int	1
fr_hdrzine(x)	int	1
fr_hdrzone(x)	int	1
fr_imidazole(x)	int	1
fr_imide(x)	int	1
fr_isocyan(x)	int	1
fr_isothiocyan(x)	int	1
fr_ketone(x)	int	1
fr_ketone_Topliss(x)	int	1
fr_lactam(x)	int	1
fr_lactone(x)	int	1
fr_methoxy(x)	int	1
fr_morpholine(x)	int	1
fr_nitrile(x)	int	1
fr_nitro(x)	int	1
fr_nitro_arom(x)	int	1
fr_nitro_arom_nonortho(x)	int	1
fr_nitroso(x)	int	1
fr_oxazole(x)	int	1

fr_oxime(x)	int	1
fr_para_hydroxylation(x)	int	1
fr_phenol(x)	int	1
fr_phenol_noOrthoHbond(x)	int	1
fr_phos_acid(x)	int	1
fr_phos_ester(x)	int	1
fr_piperdine(x)	int	1
fr_piperzine(x)	int	1
fr_priamide(x)	int	1
fr_prisulfonamd(x)	int	1
fr_pyridine(x)	int	1
fr_quatN(x)	int	1
fr_sulfide(x)	int	1
fr_sulfonamd(x)	int	1
fr_sulfone(x)	int	1
fr_term_acetylene(x)	int	1
fr_tetrazole(x)	int	1
fr_thiazole(x)	int	1
fr_thiocyan(x)	int	1
fr_thiophene(x)	int	1
fr_unbrch_alkane(x)	int	1
fr_urea(x)	int	1
rdkit.Chem.GraphDescriptors		
BalabanJ(x)	float	1
BertzCT(x)	float	1
HallKierAlpha(x)	float	1
Ipc(x)	float	1
Kappa1(x)	float	1

Kappa2(x)	float	1
Kappa3(x)	float	1
rdkit.Chem.Lipinski		
FractionCSP3(x)	float	1
HeavyAtomCount(x)	int	1
NHOHCount(x)	int	1
NOCount(x)	int	1
NumAliphaticCarbocycles(x)	int	1
NumAliphaticHeterocycles(x)	int	1
NumAliphaticRings(x)	int	1
NumAromaticCarbocycles(x)	int	1
NumAromaticHeterocycles(x)	int	1
NumAromaticRings(x)	int	1
NumHAcceptors(x)	int	1
NumHDonors(x)	int	1
NumHeteroatoms(x)	int	1
NumRotatableBonds(x)	int	1
NumSaturatedCarbocycles(x)	int	1
NumSaturatedHeterocycles(x)	int	1
NumSaturatedRings(x)	int	1
RingCount(x)	int	1
rdkit.Chem.QED		
default(x)	float	1
qed(x)	float	1
weights_max(x)	float	1
weights_mean(x)	float	1
weights_none(x)	float	1
rdkit.Chem.rdMolDescriptors		

CalcChi0n(x)	float	1
CalcChi0v(x)	float	1
CalcChi1n(x)	float	1
CalcChi1v(x)	float	1
CalcChi2n(x)	float	1
CalcChi2v(x)	float	1
CalcChi3n(x)	float	1
CalcChi3v(x)	float	1
CalcChi4n(x)	float	1
CalcChi4v(x)	float	1
CalcCrippenDescriptors(x)	tuple	2
CalcExactMolWt(x)	float	1
CalcFractionCSP3(x)	float	1
CalcHallKierAlpha(x)	float	1
CalcKappa1(x)	float	1
CalcKappa2(x)	float	1
CalcKappa3(x)	float	1
CalcLabuteASA(x)	float	1
CalcNumAliphaticCarbocycles(x)	int	1
CalcNumAliphaticHeterocycles(x)	int	1
CalcNumAliphaticRings(x)	int	1
CalcNumAmideBonds(x)	int	1
CalcNumAromaticCarbocycles(x)	int	1
CalcNumAromaticHeterocycles(x)	int	1
CalcNumAromaticRings(x)	int	1
CalcNumAtomStereoCenters(x)	int	1
CalcNumBridgeheadAtoms(x)	int	1
CalcNumHBA(x)	int	1

CalcNumHBD(x)	int	1
CalcNumHeteroatoms(x)	int	1
CalcNumHeterocycles(x)	int	1
CalcNumLipinskiHBA(x)	int	1
CalcNumLipinskiHBD(x)	int	1
CalcNumRings(x)	int	1
CalcNumRotatableBonds(x)	int	1
CalcNumSaturatedCarbocycles(x)	int	1
CalcNumSaturatedHeterocycles(x)	int	1
CalcNumSaturatedRings(x)	int	1
CalcNumSpiroAtoms(x)	int	1
CalcNumUnspecifiedAtomStereoCenters(x)	int	1
CalcTPSA(x)	float	1
GetHashedAtomPairFingerprintAsBitVect(x)	ExplicitBitVect	2048
GetHashedTopologicalTorsionFingerprintAsBitVect(x)	ExplicitBitVect	2048
GetMACCSKeysFingerprint(x)	ExplicitBitVect	167
GetMorganFingerprintAsBitVect(x, 2, nBits=124)	ExplicitBitVect	124
MQNs_(x)	array	42
PEOE_VSA_(x)	array	14
SlogP_VSA_(x)	array	12
SMR_VSA_(x)	array	10
Total:		4636

Naturally, the model itself would only operate on lower amount of features, but in the first step the most informative independent variables need to be identified. The aim was to grasp as much information as we can about each compound, in another words generate as much data as possible. A very brief research of the domain showed that features such as LogP could be correlated with the biological activity, however since the domain knowledge is insufficient to select by logical reasoning, the desire was to provide as many features as possible to the feature selection algorithms described further. The focus was on the measures that were given a compound, returned a numerical attribute, but also arrays and fingerprints that have fixed length (in order to make it possible to split them into fixed amount of different columns).

The functions that were not used either require additional arguments that could not be supplied (such as functions meant to compare two molecules) or were not able to return a valid output or an output with a fixed size.

A new DataFrame was created with columns for each of the features; The features that return arrays or ExplicitBitVects were assigned multiple columns so that each value in the array or vector is represented in a separate column. The number of columns corresponding to each feature is shown in the table above. The DataFrame had 19125 rows and 4639 columns (the 4636 feature columns from the table and the INDEX, SMILES and ACTIVE columns). Since some of the 4636 features were highly correlated and having such a high number of features will decrease the model performance, we decided to select the best features from this set.

Obviously, the number of features is very high. We could have taken a lower number of features to represent the model initially, to avoid having to deal with them in the next stage. However, since it is almost impossible for us to identify which ones have the most predictive value, this did not seem like a sensible approach. Therefore, in this step we chose the approach of “the more, the merrier”.
Alternatively, we could have left out the functions that create fingerprints, since they add a huge amount of columns (95% of all columns) that add complexity to the later feature selection. However, as we mentioned before, the added complexity is not sufficient reasoning for dropping information. Since we do not have domain knowledge, we cannot be certain if a position of certain atom at an exact place in the molecule would not highly contribute to predicting activity correctly. This was proved to be a good decision, as both feature selection methods that we employed, despite their lack of overlap, selected some fingerprint attribute in their top 20-30 features (but more on that later). We could have also not split the arrays and fingerprints into separate features. This would have made it very difficult for the algorithm to interpret them, however, since they would probably interpret fingerprint as a categorical string variable, which would limit it from recognizing underlying patterns.

Preprocessing and selecting features

Four features, MaxAbsPartialCharge, MaxPartialCharge, MinAbsPartialCharge, and MinPartialCharge, were found to contain NaN and infinite values. These features were thus dropped.

We now split the data into a train and validation set, with an 80:20 division. For this, we used a stratified split. The feature selection and training are performed on the large train set, while the validation set is used solely to evaluate our performance and generate the estimate. This means that we kept the original proportion of the labels (97.3% of 0 and 2.7% of 1) in the 2 new data sets. This was done because of the large class imbalance present in the data, which we will discuss later in the report.

There are many metrics that could be used for feature selection. Initially, we looked into using a Relief-based algorithm like ReliefF, but running this algorithm took too much time and was too computationally heavy, so we decided to settle for metrics that could be calculated more easily. We used two approaches as described below.

Approach 1: Multicollinearity and wrapper method

For the feature selection, firstly, we looked at potential **multicollinearity** in the dataset. As we suspected that a number of the features would highly correlate with each other (as many features describe similar things), pairwise correlation scores of the columns were calculated to determine which of the columns could be removed. If two features had a Pearson Correlation Coefficient higher than 0.95, one of the features was removed from the dataset.

90 features were removed, of which 51 even had a Pearson Correlation Coefficient of 1 with some other feature, i.e. they were the exact same.

Still, 4542 features remained.

Since we cannot keep that many features (otherwise the dataset would be too large for the algorithm to train considering the available computing power) we decided to reduce this number drastically by a **wrapper approach**. This method consists of trying to do the prediction using our 4542-features dataset using a fast algorithm (and obviously rather weak) in order to select a rather small subset of the more relevant features that the algorithm found. In order to do so, we have used random forests of size 10 (and default Scikit-learn configuration) together with the sequential feature selector from the Mlxtend library which performed a greedy search until it found 20 features. This number of features was arbitrarily chosen, but we felt like it was the best compromise between having a quick training and a good predictive power with enough information about the molecule.

The 20 features that were found were:

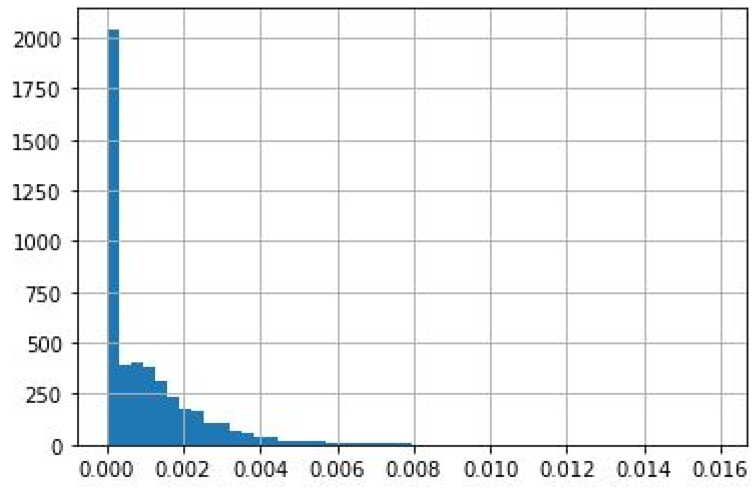
Feature
CalcNumHBD
fr_C_S
fr_hdrzone
GetHashedAtomPairFingerprintAsBitVect285
GetHashedAtomPairFingerprintAsBitVect304
GetHashedAtomPairFingerprintAsBitVect462
GetHashedAtomPairFingerprintAsBitVect1209

GetHashedAtomPairFingerprintAsBitVect1216
GetHashedAtomPairFingerprintAsBitVect1267
GetHashedTopologicalTorsionFingerprintAsBitVect88
GetHashedTopologicalTorsionFingerprintAsBitVect464
GetHashedTopologicalTorsionFingerprintAsBitVect599
GetHashedTopologicalTorsionFingerprintAsBitVect668
GetHashedTopologicalTorsionFingerprintAsBitVect719
GetHashedTopologicalTorsionFingerprintAsBitVect920
GetHashedTopologicalTorsionFingerprintAsBitVect987
GetHashedTopologicalTorsionFingerprintAsBitVect1175
GetHashedTopologicalTorsionFingerprintAsBitVect1287
GetHashedTopologicalTorsionFingerprintAsBitVect1873
GetHashedTopologicalTorsionFingerprintAsBitVect1914

Mutual Information

Next to this, we also calculated the mutual information of each of the features with respect to the label. Mutual information is a measure of the dependency between two variables and is a common metric used in feature selection.

We implemented this calculation using Scikit-Learn’s mutual_info_classif function. The mutual information score for all of our features was quite low, as can be seen in the histogram plot showing the distribution of the scores below.



We decided to place our limit at 0.07, where every feature that had a lower mutual information score than this limit was discarded. A total of 27 features remained.

The 27 features were:

Feature
CalcChi0n
CalcChi1n
CalcExactMolWt
CalcLabuteASA
CalcNumHeteroatoms
CalcTPSA
NumHDonors
fr_Ar_COO
ExactMolWt
HeavyAtomMolWt
MolWt
BalabanJ
BertzCT
GetHashedAtomPairFingerprintAsBitVect1563
MQNs_25
PEOE_VSA_0
PEOE_VSA_1
PEOE_VSA_6
PEOE_VSA_8
PEOE_VSA_9
SMR_VSA_0
SMR_VSA_4
SMR_VSA_6

SMR_VSA_9
SlogP_VSA_1
SlogP_VSA_4
SlogP_VSA_5

Interestingly enough, this list differs widely from the features found using the other approach.

Class imbalance

When we first analyzed the training set, before pre-processing the SMILES representation in the exploratory data analysis. We made a major observation that could undeniably impact the quality of the predictions. A quick look at our labels (activity) showed that the data frame is skewed: there are 18654 (97.3%) of samples labeled as 0 but only 521 (2.7%) samples labeled as 1.

What could occur in terms of prediction because of this is that a very “lazy” (or too “clever” depending on your point of view) machine learning algorithm could get away with a more than decent accuracy. Indeed, if we always predict a sample as a 0 label, we still get an accuracy of 97.3% on our data set! However, it means that 521 samples would be false negative (recall = precision = 0) which could be catastrophic if we are in fact predicting whether a molecule is carcinogenic or not and not an innocuous trait of the molecule.

As said, we had already used a stratified split in order to keep the proportions of the classes intact in our training and validation data. Then, we used an oversampling technique called SMOTE on the training set. This algorithm resolves the class imbalance by synthetic oversampling - adding fake minority data near the already existing points. After SMOTE, our training data set is composed of 50% samples labeled 0 and 50% labeled 1 which will hopefully translate into much higher values of recall and precision after validation than our “lazy” algorithm.

SMOTE seemed like a better option to us than simple copying of the data from minority class, which has multiple disadvantages that lead us to choose the more complex, synthetic oversampling method. Since in our training data (80% of the original training data) there are only 417 active molecules, in order to even out the ratio different, we would need to copy them around 35 times. This could result in a model overfitted to the specific molecules that are active, and not grasping what is similar in the active class “cluster”. SMOTE adds observations that are just similar to the original data. One worry might be that it could create compounds that do not even exist in real life - but even that is actually not an issue, if it helps the model to learn what similarities are important.
Alternatively, we could have also adjusted the cost function of each model to deal with the class imbalance by making a Type II error costly (cost-sensitive learning). However, this is generally a bit more complex to implement than SMOTE and also needs to be implemented for each algorithm.

We implemented the SMOTE function from the Imbalanced-Learn API (https://imbalanced-learn.readthedocs.io/en/stable/api.html).
Applying SMOTE increased our number of training samples from 15300 to 29766.

Summary

After using the various feature selection methods and preprocessing the data with SMOTE we ended up with the following datasets:

Dataset	Feature Selection Method	Number of features
1	PCC + decision tree wrapper method	20
2	Mutual Information	27

There were 29766 samples in the training set and 3825 samples in the validation set.

Modeling

Because the time constraint did not allow us to do much experimentation, and after failing with the implementation of the promising Hellinger distance decision tree (HDDT), we opted for two boosted decision tree algorithms. Ensemble learning is a very popular and successful approach (especially the second algorithm that we used, XGBoost, is popular at the moment). The algorithm selection had the constraints of training and tuning time (which in the case of neural networks for example, would be expected to be higher. This, thanks to the versatility of applications that boosting algorithms are successful in resulted in our decision to apply two of them.

The first algorithm: AdaBoost

The first algorithm we used is AdaBoost, a famous boosting algorithm that is easier to train than XGBoost, the second algorithm which we will discuss in a second. We used a decision tree algorithm as a base learner.

We implemented AdaBoost using Scikit-Learn, with the following parameters:

base_estimator=DecisionTreeRegressor(criterion='mse',
max_depth=7,
max_features=None,
max_leaf_nodes=None,
min_impurity_decrease=0.0,
min_impurity_split=None,
min_samples_leaf=1,
min_samples_split=2,
min_weight_fraction_leaf=0.0,
presort=False,
random_state=None,

splitter='best'),
learning_rate=1,
loss='linear',
n_estimators=100,
random_state=None)

The second algorithm: XGBoost

The other algorithm chosen is XGBoost. Mainly because it has the reputation of being a Kaggle competition “serial winner”. Indeed, its versatility on a very large panel of data science problems, including similar binary classification problem, and its straightforward implementation using Scikit-learn, make XGBoost one of the best first candidate to try in those circumstances.

Since XGBoost is a tunable algorithm with a lot of hyperparameters we could not use brute force search on every parameter to determine the best configuration as it would be too computationally demanding. Instead, we started by choosing a relatively high learning rate and we used the cross-validation function from the XGBoost library to get the optimal number of estimators (we used a tree-based model as mentioned earlier). Secondly, using the optimal values that we identified, we performed a grid search (using GridSearchCV function from scikit-learn) on the max_depth and min_child_weight parameters as they would have the highest impact on overfitting and then we did the same with gamma, subsample, colsample_bytree and the L1 regularization parameter reg_alpha. Finally, we decreased the learning rate and apply the cross-validation function one more time to get a new number of estimators. We kept the configuration that has given us the highest AUC-score on the validation set.

Each simulation was performed using a stratified 10 fold cross-validation.

Thanks to the previous method we ended up using the following configurations for the xgboost algorithm:

- Dataset 1:
  objective = ‘binary:logistic’
  learning_rate = 0.1
  n_estimators = 421
  max_depth = 6
  min_child_weight = 1
  gamma = 0
  subsample = 0.8
  colsample_bytree = 0.8
  reg_alpha = 0

- Dataset 2:
  objective = ‘binary:logistic’
  learning_rate = 0.1

n_estimators = 662
max_depth = 4
min_child_weight = 1
gamma = 0
subsample = 0.6
colsample_bytree = 0.7
reg_alpha = 0

Evaluation

As stated earlier, we divided the original training dataset into a training and validation set with the ratio 80:20. The model performances will be evaluated based on the AUC-scores that these models, trained on the training set, obtain on this validation set. The score on the validation set will be used as our AUC-score estimate for the test set as well.

Results

After running and tuning our models we got the following AUC-scores on the validation set:

	AdaBoost	**XGBoost**
Dataset 1	0.7355	0.6889
Dataset 2	0.7565	0.7711

The winner, in this case, is the XGBoost algorithm on the second dataset (the one based on Mutual Information). This was this particular combination of features and algorithm that was picked for the prediction of the test set.