Automatic Patent Searching via Extreme Multi-Label Classification

Conor McCabe

Mansfield College

University of Oxford

Submitted in partial fulfilment of the degree of

Master of Science in Statistical Science

September 2019

For my parents

Acknowledgements

I am deeply appreciative of my supervisors Prof. Garrett Morris and Dr. Willem van Hoorn for their patience and guidance throughout this project. They gave extremely generously of their time and were enthusiastic and engaged from beginning to end.

Abstract

In this dissertation, I attempt to answer the question “Can computers replace patent attorneys?”. In order to explore this problem, I compare models for classifying compounds according to the patents under which they are registered. An existing Naïve Bayes approach is reproduced using open source software and it is confirmed that the problem is amenable to modelling. The problem is reformulated as one of multi-label classification and I compare the accuracy of the Sparse Local Embeddings for Extreme Multi-label Classificatrion (SLEEC) algorithm, a state-of-the-art embedding-based model, with an ensemble of Random Forest and Extremely Randomised Trees models, where the base models are stacked using a majority voting procedure and hyperparameters are tuned using Bayesian optimisation. In a final comparison, the ensemble approach is shown to outperform both the Naïve Bayes model and SLEEC.

Contents

List of Figures

List of Abbreviations

SMILES	Simplified Molecular Input Line Entry System.
ECFP	Extended Connectivity Fingerprint.
CNN	Convolutional Neural Network.
RNN	Recurrent Neural Network.
VAE	Variational Autoencoder.
IUPAC	International Union for Pure and Applied Chemistry.
OPSIN	Open Parser for Systematic IUPAC Nomenclature.
RF	Random Forest.
ERT	Extremely Randomised Trees
KNN	K-Nearest Neighbours.
QSAR	Quantitative Structure-Activity Relationship.
QSPR	Quantitative Structure-Property Relationship.
IPA	Intellectual Patent Analytics.
SVD	Singular Value Decomposition.
XML	Extreme Multi-Label Classification.
SLEEC	Sparse Local Embeddings for Extreme Multi-Label Classification.
SVP	Singular Value Projection.
ALS	Alternating Least Squares.
ADMM	Alternating Direction Method of Multipliers.
MLRF	Multi-Label Random Forest.
LPSR	Label Partitioning for Sublinear Ranking.
nDCG	Normalised Discounted Cumulative Gain.
BR	Binary Relevance.
CC	Classifier Chain.
LP	Label Powerset.

CSR	Compressed Sparse Row.
TSVD	Truncated Singular Value Decomposition.
BO	Bayesian Optimisation.
GP	Gaussian Process.

Man is a centaur, a tangle of flesh and mind,
divine inspiration and dust.

— Primo Levi [1]

1 Introduction

Contents

1.1 Motivation

As Asimov's Three Laws of Robotics written in 1950 attest [2], anxieties about the rise of machine intelligence have loomed in the human psyche since the inception of machines themselves. Recent advances in machine learning applications have seen such anxieties shift from doomsday scenarios to more mundane worries about the changing nature of work in an increasingly automated world. With up to 47% of US jobs purportedly at risk of automation [3], such fears are not unfounded. There exists however a more optimistic vision of the future of humans and machines; of a symbiotic relationship in which AI serves to amplify and enhance human creativity rather than replace it entirely.

Chess world champion Garry Kasparov's infamous 1997 loss to IBM's Deep Blue [4] represents perhaps the height of popular conceptions of AI as a harbinger of humanity's demise. More recently though, an augmented intelligence, or "centaur", approach to chess has emerged which pairs human players with chess programs and has proven more effective than machines acting alone [5]. This centaur approach has found ready application in the field of active learning, in which algorithms are designed that can query the user in order to improve performance [6].

Exscientia is a drug discovery firm which has pioneered the centaur approach to drug discovery, where human decision making is enhanced rather then replaced by AI systems. A current bottleneck in the process of bringing a new drug to market is the slow, manual process of determining whether the candidate molecule might be protected under existing patents [7]. "Can computers replace patent attorneys?" is the question which inspired this project, and one which I will attempt to answer.

Aligning with the centaur approach to problem solving, the goal of this project is not to eliminate entirely the need for the subjective evaluation of patents, but rather to provide information so that human actors can make decisions informed to the greatest possible extent. Accordingly, I will aim to build a model which reduces the space of patents that need to be searched from *all* of those currently registered, to a subset of patents which could then be examined to a more rigorous degree. This would vastly reduce the time spent searching unrelated patents and would enable resources to be efficiently allocated towards more promising routes of enquiry.

In the remainder of this chapter, I introduce the requisite chemoinformatic knowledge which was used throughout the project and describe the chemical databases from which the data used to train the model was extracted. I review the literature of machine learning methods applied to the field of drug discovery and chemoinformatics more generally, and finally, I reformulate the problem as one of extreme multi-label classification and review the techniques used to tackle problems of such scale.

Chapter 2 discusses the methods used to fit and tune our final model, as well as detailing the required data cleaning and preprocessing. In Chapter 3, I examine the existing work in classifying compounds by patent and reproduce a proprietary implementation of a Naïve Bayes model using Python. Chapter 4 describes novel approaches taken to solving the problem, in which I discuss the process of fitting both an existing embedding approach and a heuristically built model ensemble. I further detail the methods used to tune hyperparameters and finally compare the performance of both methods with the Naïve Bayes model outlined in Chapter 3. Lastly in Chapter 5, I offer a summary and conclusion as well as suggesting areas of further work on the topic.

1.2 Patenting Drugs

In order to develop a new drug, one must not simply discover the new compound, but must also determine whether it is *sufficiently* novel and whether it infringes on any patents covering existing compounds [8]. This is not a trivial problem and currently involves a slow manual search by a patent attorney to assess the novelty of the drug in question.

As can be seen in Figure 1.1, the notion of sufficient novelty can be difficult to determine. Sildenafil, the compound on the left, was patented before Vardenafil, the compound on the right, and so producers of Vardenafil had to prove sufficient novelty of their new molecule. It transpired that the change in the outer bond (circled in red) was not a significant enough difference to justify a new patent, but the new ring system (circled in blue) represented sufficient novelty of chemical structure to allow for a separate patent to be granted.

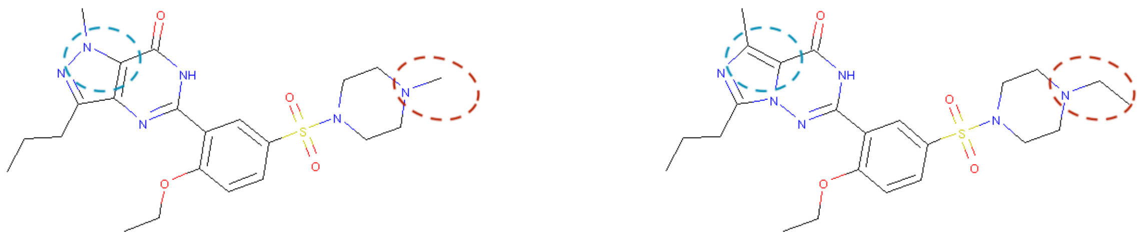
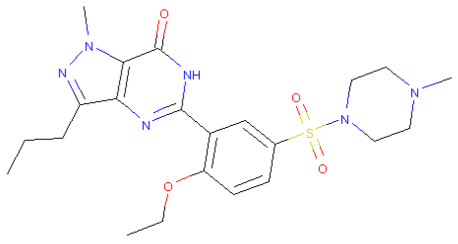


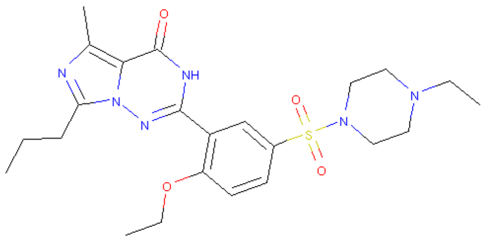
Figure 1.1: Sildenafil (left) and Vardenafil (right). Pfizer owns the patent for Sildenafil, but Bayer Pharmaceuticals, GlaxoSmithKline, and Schering-Plough were granted a patent for Vardenafil based on the new ring system circled in blue.

1.3 Representing Molecules in Computers

Chemical structures are most commonly stored in computers as a molecular graph, with nodes corresponding to the atoms of the molecule and edges corresponding to the bonds connecting them [9]. One way to represent and communicate a molecular graph is via linear notation; an encoding of the molecular structure using alphanumeric characters. A widely used linear notation is Simplified Molecular Input Line Entry Specification (SMILES) [10]. SMILES represents atoms by their atomic symbol (e.g. O for oxygen), includes information about bond type (= for double, # for triple, with single bonds implicitly assumed), and uses a hydrogen suppressed notation (hydrogen atoms are ordinarily omitted). An example of a two dimensional graphical representation of the molecules Sildenafil and Vardenafil alongside their SMILES strings can be seen in Figure 1.2.



(a) CCCc1nn(C)c2C(=O)NC(=Nc12)c3cc(ccc3OCC)S(=O)(=O)N4CCN(C)CC4



(b) CCCc1nc(C)c2C(=O)N=C(Nn12)c3cc(ccc3OCC)S(=O)(=O)N4CCN(CC)CC4

Figure 1.2: Sildenafil (a) and Vardenafil (b) alongside their canonical SMILES strings.

Since there are many possible SMILES strings for a given molecule, it is important to establish a canonical representation. A widely used algorithm for determining canonical order is the Morgan algorithm [11], which iteratively computes the connectivity (the number of other atoms to which an atom is connected) of each atom in order to effectively differentiate between them. Additional chemical properties such as bond order and atomic number are considered if a tie occurs between two atoms with equal connectivity values. Critically, the Morgan algorithm produces its ordering independently of the original numbering of the atoms. Once a canonical ordering has been produced, a unique SMILES string can then easily be generated. The Morgan algorithm can be used to generate binary fingerprints that describe the chemical substructure of a molecule.

A descendant of the Morgan algorithm is the Extended-Connectivity Fingerprint (ECFP) [12], a class of topological fingerprints used for molecular characterisation. The ECFP algorithm was specifically designed to capture features related to molecular activity and has been optimised for tasks involving the prediction of drug activity.

The ECFP generation process has three stages:

1. An initial assignment stage in which each atom is assigned an identifier.
2. An iterative updating stage in which each atom identifier is updated to reflect the identifiers of the atom's neighbors, including identification of whether it is a structural duplicate of other features.
3. A duplicate identifier removal stage in which multiple occurrences of the same feature are reduced to a single representative in the final feature list.

The unique identifiers generated by this process correspond to the chemical substructures contained within the full molecule. These unique identifiers will be used later as the features of the machine learning model.

Generating ECFP requires setting a diameter parameter, i.e. considering all substructures contained within the molecule up to a certain size.

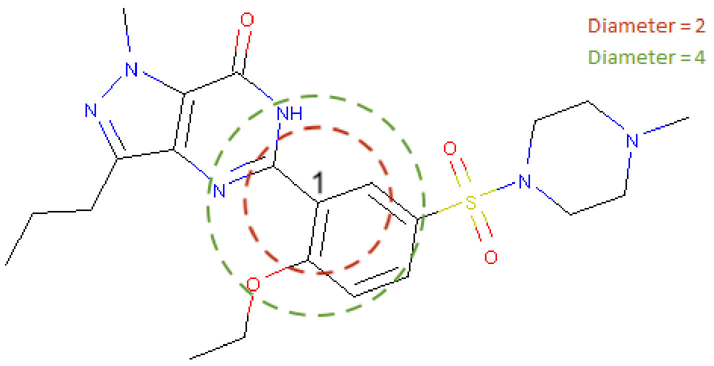


Figure 1.3: Illustration of substructures of different diameters.

Figure 1.3 illustrates the concept of molecular diameter, D. Starting at the atom marked by the 1 at the centre of the figure, substructures within $D = 2$ are all those which can be constructed using atoms which are directly connected to the initial atom and are enclosed above by the red dotted line. Similarly, substructures of diameter $D = 4$ are all structures which can be constructed using atoms within two connections of our original atom at 1 and are enclosed by the green dotted line.

1.4 Data

Two sets of compounds extracted from registered patents are considered in this project - SureChEMBL and NextMove.

SureChEMBL is a publicly available large-scale data set containing annotated molecules extracted from the full text, images, and attachments of patent documents [13]. Since molecular structure is automatically extracted from both text and images, there is the possibility that some molecular representations are incorrect. The SureChEMBL set contains approximately 20 million chemical structures abstracted from 1.7 million patents.

The second data set used in this project was developed by NextMove Software and consists of chemical interaction data extracted from US patent grants from the period 1976-2013, and US patent applications covering the period 2001-2013 [14]. The NextMove data set was created using OPSIN (Open Parser for Systematic IUPAC Nomenclature), an open source algorithm which parses patents and converts chemical names mentioned within to their SMILES representations [15]. The NextMove set additionally extracts chemical reactions data from within the text and stores the data in reaction SMILES form [16]. Reaction SMILES are strings of the form

$$\begin{gathered}\text{CC(=O)O.OCC} > \text{[H+].[Cl-].OCC} > \text{CC(=O)OCC} \\ \text{Reactant} > \text{Agent} > \text{Product}\end{gathered}$$

and provide information about the reagents required to synthesize the final compound.

This is useful because only the final chemical compound produced by the reaction is likely to be of relevance to the patent application and hence any agents and reactants used to generate the final compound can be ignored. The NextMove set covers 135 thousand patents and contains 1.08 million unique compounds.

1.5 Machine Learning and Drug Discovery

An oft cited number for the cost of developing and bringing a drug to market is \$2.6 billion [17], and as such small increases in efficiency can prove extremely lucrative. Where previously physical models requiring explicit equations based on known chemical relations were used, over the past two decades machine learning methods have begun to replace physical models in drug discovery and chemoinformatics [18].

The past decade in particular has seen a remarkable increase in both the quality and quantity of biological data available [19], which has meant that machine learning methods such as Deep Learning, requiring vast amounts of data in order to perform well, have found increasing use in the area of drug discovery [20]. A small sample of machine learning methods used for drug discovery are given below.

Convolutional Neural Networks (CNNs) have been used to generate molecular fingerprints inspired by the Morgan fingerprints described above which have been shown to perform better than standard fingerprints on certain predictive tasks [21]. SMILES strings have been used as the input for Recurrent Neural Networks (RNNs) whereby rather than taking just a single SMILES string (e.g. the canonical representation), the author of the paper augmented their data set with many different SMILES representations of the same molecule, which surprisingly led to an increase in performance [22].

Variational Autoencoders (VAEs) have been used to map molecules into continuous latent spaces where novel molecules with particular properties can be generated by perturbing known chemical structures or by interpolating between two molecules with desirable properties [23]. In order to overcome the problem of the model outputting strings that were not valid molecules, a new VAE architecture has been developed to allow for the explicit encoding of the SMILES grammar to ensure that all outputs were valid SMILES strings [24].

A wide range of machine learning models have found use in the field of Chemoinformatics more generally, including Random Forest and Logistic Regression to model the bioactivity of ligands [25], Random Forest for the prediction of human cytochrome inhibition [26] and for Quantitative Structure–Activity Relationship (QSAR) modelling [27], K-Nearest Neighbours (KNN) for modelling Quantitative Structure–Property Relationships (QSPR) of metabolic stability [28] and for ligand-based modelling [29], Naïve Bayes for classifying kinase inhibitors [30], and Bayesian Optimisation for generating low-energy molecular conformers [31].

Separately, there has been an increased use in recent years in applying machine learning techniques within the sphere of Intellectual Patent Analytics (IPA) [32]. IPA is the process of analysing patent data in order to extract usable insights. Naïve Bayes has been used to classify patents based on metadata and citation information [33] and Singular Value Decomposition (SVD) and K-Means clustering have been used to cluster patent data [34].

More recently, an algorithm has been developed that can generate novel syntheses of existing drugs by avoiding those covered under existing patents [35]. It works by substituting unconventional yet chemically feasible synthesis routes for patented portions of the production process, which it draws from a wide database of chemical reactions.

1.6 eXtreme Multi-Label Classification

Rather than considering the problem as a multi-class classification problem whereby each molecule is predicted to belong to exactly one of a number of mutually exclusive classes, a more flexible formulation of the problem is one of *multi-label* classification.

Multi-label classification is a supervised learning paradigm in which a particular data point is allowed to simultaneously be part of multiple classes [36]. As a framework, allowing the model to suggest multiple similar patents under which it believes a new compound might be covered is more desirable than suggesting a single most likely patent, since the goal of this project is only to reduce the space of patents which need to be searched.

Given the size of the data sets, ready comparisons can be made to the problem of eXtreme Multi-Label classification (XML), a well established problem in the field. The purpose of XML is to train a model to tag a new data point with a subset of labels from a large label set where the number of possible labels can be in the hundreds of thousands or even millions [37]. A benchmark data set in the field is the Wikipedia document classification problem where the number of labels exceeds one million, similar in scope to the problem of assigning molecules to a subset of over a million patents. The two most popular approaches to XML classification are embedding-based approaches [37–39] and using tree-based learners [40, 41].

1.6.1 Embedding Methods

One way to overcome the huge size of the label space is to project it into a lower dimensional space, thereby reducing the number of effective labels. Given a set of n training points $\{(\mathrm{x}_i, \mathrm{y}_i)\}_{i=1}^{n}$ with d-dimensional feature vectors $\mathrm{x}_i \in \mathbb{R}^d$ and L-dimensional label vectors $\mathrm{y}_i \in \{0,1\}^L$, embedding approaches project the label vectors onto a linear subspace of dimension $\hat{L}$ such that $\mathrm{z}_i = \mathbf{U}\mathrm{y}_i$. We can then train regressors to predict z_i as $\mathbf{V}\mathrm{x}_i$ in the lower dimensional label space before projecting back to the full space so that $\mathrm{y} = \mathrm{U}^*\mathrm{V}\mathrm{x}$, where U^* is a decompression matrix which lifts the embedded vectors back to the full label space.

A number of different embedding methods have been successfully utilised including compressed sensing [42], Bloom filters [43], and Singular Value Decomposition [44].

SLEEC

The current state-of-the-art embedding approach is Sparse Local Embeddings for Extreme Multi-label Classification (SLEEC) [37], which differs from existing methods in a number of key ways.

Firstly, rather than projecting labels globally onto a linear low-rank space, SLEEC non-linearly learns embeddings which preserve the pairwise distance between only the closest (as opposed to all) label vectors, thus capturing label correlations. That is $d(\mathrm{z}_i, \mathrm{z}_j) \approx d(\mathrm{y}_i, \mathrm{y}_j)$ only if $i \in \mathrm{KNN}(j)$ where d is a distance metric. The SLEEC method makes use of Singular Value Projection (SVP) [45] to learn the embeddings, which preserves the nearest neighbours in the embedded space. For prediction, rather than using a decompression matrix, SLEEC uses a KNN classifier *within* the subspace, leveraging the fact that the nearest neighbours have been preserved during training. Thus, for a novel point $\tilde{x}$, the predicted label is obtained by

$$\tilde{\mathrm{y}} = \textstyle\sum_{i:\mathbf{V}x_i \in \mathrm{KNN}(\mathbf{V}\tilde{x})} \mathrm{y}_i.$$

In order to expedite training times, SLEEC clusters the training data, learns a separate embedding per cluster and performs KNN classification only within the test point's cluster. To tackle possible instabilities associated with clustering in very large dimensions, SLEEC makes use of an ensemble in which each learner is generated by a different clustering.[1]

1.6.2 Tree Based Methods

Many decision tree-based methods have found success in the area of XML. The Multi-Label Random Forest (MLRF) [40] method learns an ensemble of randomised trees and efficiently extends the multi-class Random Forest classifier to the multi-label case by assuming independence of labels. The Label Partitioning for Sublinear Ranking (LPSR) [46] approach learns a label hierarchy which reduces the training time of base classifiers from linear in the number of labels to sublinear, allowing for efficient scaling for large numbers of labels.

[1]The SLEEC algorithm is examined in more detail in Section 2.3.2.

The current state-of-the-art tree-based method is FastXML [41], whose primary technical contribution is to replace the expensive entropy and GINI index based node partitioning formulations of Random Forest and instead propose a more efficient node partition which instead optimises the normalised Discounted Cumulative Gain (nDCG) [47], a measure which is sensitive to ranking and thus ensures positive labels are predicted with ranks as high as possible.

1.6.3 Problem Transformation Approaches

Though not as popular in the eXtreme Multi-label Classification case, problem transformation approaches have been used extensively in smaller scale multi-label classification tasks [48–50].

The Binary Relevance (BR) [48] approach is a subset of the one-vs-all paradigm and involves training a separate learner for each label, transforming the multi-label classification problem into L binary classification problems, where L is the number of labels. Two main problems with this method exist in relation to our problem; firstly, there is an assumption of independence between the labels which is unlikely to be true in the case of drug patents (e.g. patents containing drugs addressing a certain medical issue are likely to be related) as well as the fact that it is computationally unfeasible for large numbers of labels.

Classifier Chain (CC) [48] methods involve generating q binary learners (where $q < L$, the full label dimension) where each subsequent learner incorporates the predictions made by previous classifiers in the chain as additional features. While taking account of label correlations, this method still suffers from the issue of scalability to large label spaces where thousands of individual learners would still need to be trained.

The Label Powerset (LP) [50] approach involves treating each unique combination of labels as its own class, thus transforming the problem to standard multi-class classification. This has the attractive quality that only a single multi-class classifier needs to be trained, however it is prone to underfitting for large label spaces [51]. Additionally, all three methods above can perform poorly in the the presence of imbalanced distributions of class labels [52].

As can be seen in Figure 1.4, the data is made up of a large number of tail labels. Tail labels are those which occur only infrequently throughout the data (often defined as occurring in fewer than five data points [37]) and make up 60.1% of the

NextMove set.[2] It may therefore be wise to choose a classification method which is robust when trained on a heavy-tailed label distribution.

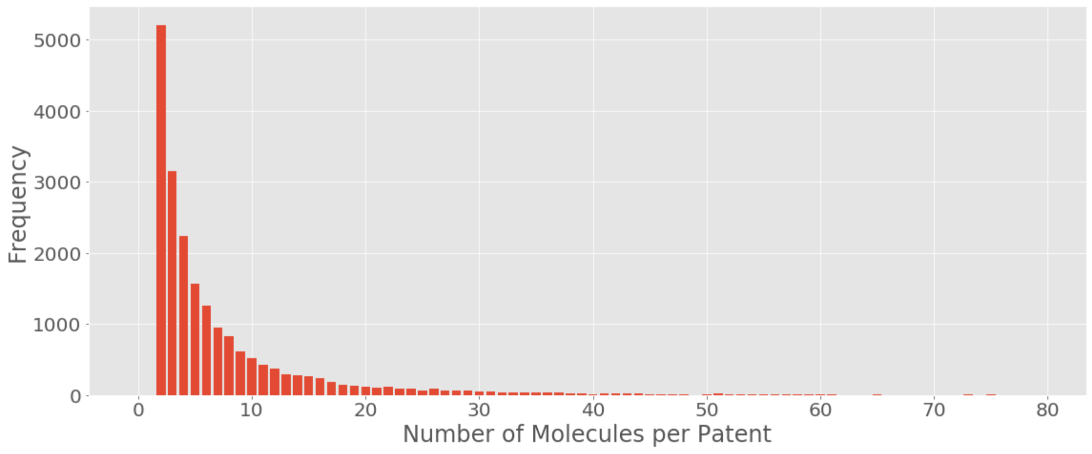


Figure 1.4: Plot of the distribution of labels. Most patents contain only a small number of molecules.

The table below gives the computational complexity of each of the problem transformation approaches detailed above.

Problem Transformation	Computational Complexity
Binary Relevance, BR	$\mathcal{O}(N_L \times \texttt{(BCC)})$
Clasiffier Chain, CC	$\mathcal{O}(q \times \texttt{(BCC)})$
Label Powerset, LP	$\mathcal{O}(\texttt{BCC})$

Above, N_L is the number of labels and `BCC` is the Base Classifier Complexity.

It is clear that since most methods scale linearly in the number of labels, they will not be computationally feasible for a problem of our size. Since the LP approach requires only a single classifier and scales only in the number of label *combinations*, this is likely to be the most scalable approach.

1.7 Software

The majority of the analysis of this project was carried out using Python, with some of the data preprocessing also performed using R. Machine learning models were trained using the `scikit-learn` [53] Python package, and problem transformation approaches were implemented using the `scikit-multilearn` package [51]. The `RDKit` package [54] was used to generate chemical fingerprints[3], and Bayesian

[2]In the case of patent classification, a "tail" patent is one which contains only a small number of molecules.

[3]Discussed in more detail in Section 2.2.1.

Optimisation was carried out using the `GPyOpt` package [55].

Truth ... is much too complicated to allow anything but approximations.

— John von Neumann [56]

2 Methods

Contents

2.1 Introduction

In this chapter, I describe the methods used to fit and tune a final classification model. I offer a description of the data used as well as the preprocessing which was carried out. I further detail the base models which were considered as well as the model stacking and ensemble approaches used to enhance classification accuracy. Finally, I outline the hyperparameter tuning process in which a Bayesian Optimisation of the hyperparameters of the base models is performed.

2.2 Data

In order to deal with the scale of the problem using the resources available, the NextMove set is randomly partitioned into ten subsets and analysis is exclusively carried on a single subset. Since the number of features of the data scales almost linearly with the label size (as can be seen in Figure 2.1), this serves to reduce not only the label space but also the feature space.

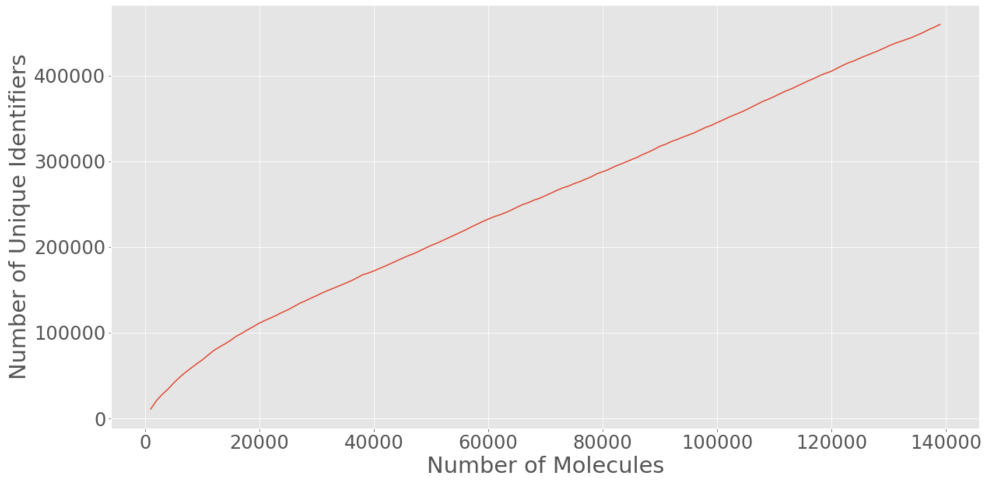


Figure 2.1: Plot showing the growth of the unique chemical substructure identifiers with the number of molecules.

Such a partition will result in approximately 10-30 patents suggested by the model as being likely to cover a test compound, which is readily searchable in a manual fashion. In partitioning the data, there is a trade off between the increased accuracy gained by increasing the number of partitions, and the added workload of manually searching a larger number of patents.

2.2.1 Preprocessing

Both the NextMove and SureChEMBML sets contain molecules stored in SMILES representations. From these SMILES strings, the unique chemical substructure identifiers corresponding to the atoms and functional groups of the molecules will need to be extracted, which will then be used as the features of the machine learning model. To accomplish this, I used the `RDKit` package in Python.

`RDKit` is a collection of chemoinformatics libraries which can be used to manipulate and extract information from molecular structures [54]. For the SureChEMBL set, I used `RDKit` to generate Morgan fingerprints from SMILES strings as described in

Section 1.3. In generating fingerprints, `RDKit` uses connectivity information similar to the process of generating ECFP, with the small difference that ECFP is defined by a diameter whereas `RDKit`'s circular fingerprints take a radius argument as their input. I opted for an ECFP diameter of six (corresponding to `RDKit` fingerprint radius 3) and generate fingerprints with a bit size of 1024.

Having generated the fingerprint for a given compound, `RDKit` was then used to extract the unique identifiers corresponding to the substructures present in the molecule. A binary data matrix $\mathbf{X}$ was generated, where the (i, j) entry of $\mathbf{X}$ contains a 1 if substructure j is present in molecule i and a 0 if it is not. Similarly, I created a label matrix $\mathbf{y}$ whose columns correspond to the patents in our data and whose rows represent the compounds, where the (i, j) entry of $\mathbf{y}$ is set to 1 if molecule i is listed as part of patent j, and 0 if it is not.

In addition to the preprocessing of the SureChEMBL set described above, the NextMove set requires an additional cleaning step. As described in Section 1.5, rather than listing a single SMILES string per data point NextMove provides reaction SMILES [16], which gives a description of the molecules used in a chemical reaction. An example of a reaction SMILES string is given in Figure 2.2.

CC(=O)O.OCC > [H+].[Cl-].OCC > CC(=O)OCC

Reactant > Agent > Product

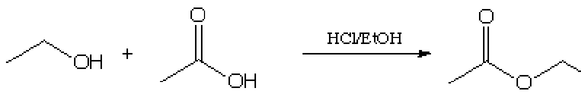


Figure 2.2: Reaction SMILES string for an example chemical reaction. The reactant (left) and agent (middle) are used to synthesize the product (right). Example taken from [16].

Since reactants and agents are unlikely to form part of the set of compounds protected under a patent, adding these to the data will introduce unnecessary noise to the model. I therefore wrote a script to extract only SMILES strings to the right of the last ">" and only use the final products as inputs in the model. SureChEMBL does not discriminate between reactants, agents, and products in this way, and so there is likely to be significantly more noise in a model trained on the SureChEMBL set compared to one trained on the NextMove data. For this reason, SureChEMBL is only used in Chapter 3 to reproduce the existing work on the topic and not in Chapter 4 when investigating novel approaches.

2.2.2 Train-Test Split

For a balanced split of the data into train and test sets, any patent which occurs in the test set must previously have been observed in the training set. Any patent which contains only a single molecule must therefore be removed from the data.

2.2.3 Data Sparsity

I now examine the structure of a single partition of the NextMove set, which will form the data used as part of implementing novel approaches in Chapter 4. The data consists of an $n \times L$ label matrix $\mathbf{y}$ and an $n \times p$ data matrix $\mathbf{X}$, where $n = 117,861$ molecules, $L = 20,411$ patents, and $p = 479,611$ features. Due to memory issues, these matrices cannot be stored in conventional form and so a sparse matrix representation is used. Both label and data matrices are extremely sparse, with a label matrix density of only 0.009% and a data matrix density of only 0.01%, hence utilising sparse matrix storage will reduce memory usage considerably. Figure 2.3 illustrates the sparsity of the first 400 rows and 800 columns of the data matrix, where red dots indicate non-zero entries.

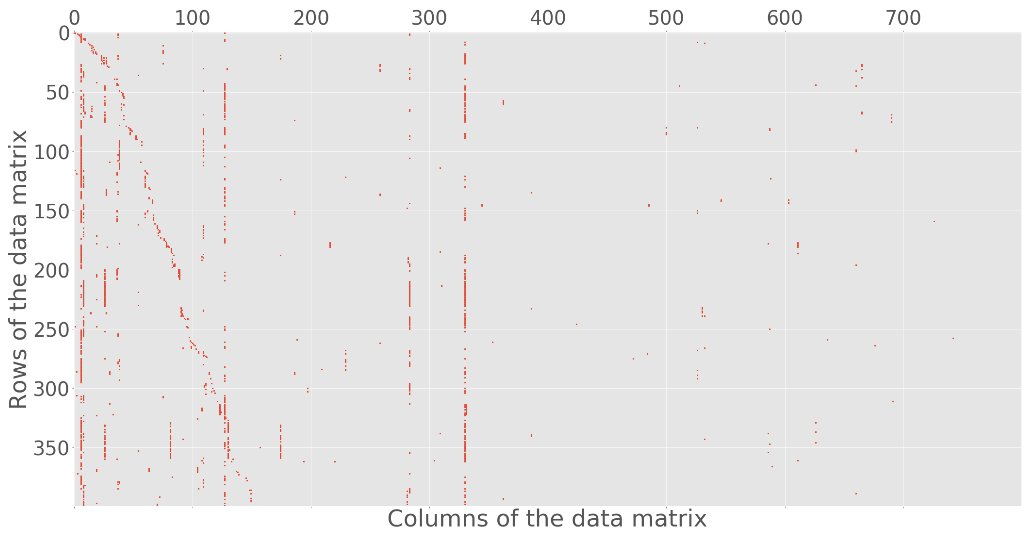


Figure 2.3: Sparsity plot of the the first 400 rows and 800 columns of the data matrix. Red dots correspond to non-zero entries of the matrix.

Compressed Sparse Row (CSR) matrix storage format is used, which stores the matrix in three arrays:

1. An array containing the non-zero entries of the matrix.
2. An array containing the column indices of the non-zero entries of the matrix.

3. An array whose i^{th} entry equals the number of non-zero entries up to and including the i^{th} row of the original matrix.

As an example, the matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

would be stored in the following arrays:

$$[\ 5\ 8\ 3\ 6\ ] \quad [\ 0\ 1\ 2\ 1\ ] \quad [\ 0\ 2\ 3\ 4\ ]$$

Storage is saved when using CSR format for an $m \times n$ matrix when the number of non-zero entries is less than $(m(n-1)-1)/2$. The number of non-zero entries of the data matrix (5.7×10^6) is far lower than the upper limit required to ensure memory savings (2.6×10^{10}) and so storing the data in this form results in a significant reduction in memory.

Additionally, sparse matrix storage can also lead to greater computational efficiency. For example, certain sparse matrix multiplication algorithms have been shown to be more computationally efficient than leading dense implementations subject to certain conditions on the sparsity of the matrix [57].

2.2.4 Dimensionality Reduction

Due to the large feature space, I performed dimensionality reduction using Truncated Singular Value Decomposition (TSVD) [58]. TSVD works by taking only the k largest singular values of the singular value decomposition, vastly decreasing compute time for large matrices. Crucially, TSVD does not require centring of the data and so the sparsity of the data matrix is preserved, further reducing computation time. It has been proven that TSVD provides the best possible rank k approximation of the full matrix [59]. The `scikit-learn` `TruncatedSVD` function is used to implement the dimensionality reduction and the results are plotted in Figure 2.4. Even taking the first 500 dimensions only leads to an explanation of 53.6% of the variance in the data. Additionally, it was deduced experimentally that training times for most base learners increased significantly after applying dimensionality reduction due to the loss of the extreme sparsity of the original data matrix. Given that there is both a loss of information *and* an increase in compute time, I decided to proceed without a reduction in the dimensionality of the feature space.

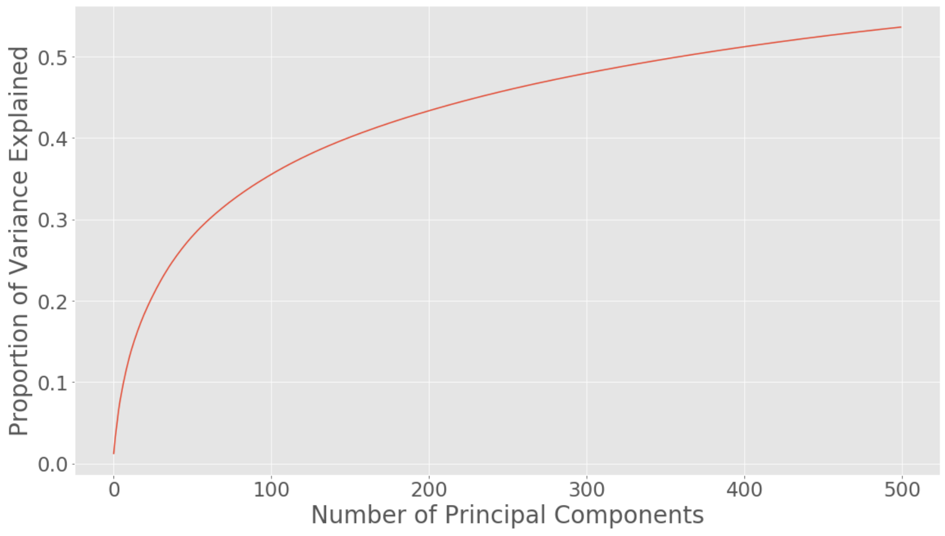


Figure 2.4: Plot of the first 500 principal components against total variance explained.

2.3 Models

In this section I discuss the different types of model which were fit to the data as part of this project. I discuss the Naïve Bayes model used in Chapter 3, as well as the SLEEC and tree-based ensemble used in Chapter 4, before finally describing Bayesian Optimisation, the method used to tune hyperparameters of the final model ensemble.

2.3.1 Naïve Bayes Classifier

Naïve Bayes is a generative model based on a simple assumption; that the features of the data are conditionally independent given the class labels [60]. A generative modelling approach seeks to first model the joint probability distribution $p(X, y)$ of the data X and labels y before using Bayes' Rule to determine the posterior distribution $p(y|X)$ and outputting the most likely class label as its model prediction. This is in contrast to discriminative learning which models $p(y|X)$ directly.

Despite Vapnik's [61] caution in *"Statistical Learning Theory"* that "one should solve the problem directly and never solve a more general problem as an intermediate step", Naïve Bayes has been shown to perform competitively as a classifier [62]. Ng and Jordan [60] posit that while discriminative classifiers have lower asymptotic error than generative models, the limits are approached much faster in the generative learning case in certain situations.

The Naïve Bayes model involves decomposing the posterior $p(y|X)$ as

$$p(y|X) = \frac{p(X,y)}{p(X)}$$

where $p(X,y)$ is the joint distribution of the data and labels, and $p(X)$ is the marginal likelihood of the data.

Since the marginal likelihood is only a function of the data, it is constant for all classes and we can thus restrict our interest to the joint probability $p(X,y)$, which we can expand as $p(X|y)p(y)$, the likelihood of our data multiplied by the prior probability of the labels.

If our data X consists of p feature vectors $X_1,\dots,X_p$, then using our assumption of conditional independence given the labels we can write:

$$p(X|y) = p(X_1,\dots,X_p|y) = \prod_{j=1}^{p} p(X_j|y).$$

Our classification rule is then:

$$\arg\max_y p(y|X) = \arg\max_y p(y)\prod_{j=1}^{p} p(X_j|y).$$

We further need to specify the likelihood model for our data, $p(X|y)$. Some common choices include a Gaussian likelihood for continuous data and a Bernoulli distribution for binary data [63]. Given that we are dealing with binary data, we will use a Bernoulli Naïve Bayes model.

Now assume we have data $\{(x_i,y_i)\}_{i=1}^{n}$ with $x_i \in \{0,1\}^p$ and $y_i \in \{1,\dots,k\}$ generated from a joint probability distribution $p(X,y)$. If we let the probability of belonging to class k, $p(y=k)=\phi_k$ for $k=1,\dots,L$, then our generative model for a single point i becomes [64]:

$$\begin{aligned} p(x_i,y_i|\phi,\pi) &= p(y_i|\phi)\prod_{j=1}^{p} p(x_{ij}|y_i,\pi) \\ &= \prod_{k=1}^{L}\phi_k^{\mathbb{I}(y_i=k)}\prod_{k=1}^{L}\prod_{j=1}^{p}[\pi_{jk}^{x_{ij}}(1-\pi_{jk})^{1-x_{ij}}]^{\mathbb{I}(y_i=k)}, \end{aligned}$$

where π_{jk} is the probability of feature j being present in class k (or in our case, the probability of patent k containing molecular substructure j) and x_{ij} is a Boolean corresponding to feature j being present in data point i (i.e. molecule i containing substructure j).

We can find estimates for parameters π_{jk} and ϕ_k by maximising the log likelihood for the data. Assuming the data are independent and identically distributed, we can write the log likelihood [64]:

$$\ell(\pi, \phi; X, y) = \sum_{k=1}^{L} n_k \log\, \phi_k + \sum_{j=1}^{p} \sum_{k=1}^{L} \sum_{i:y_i=k} x_{ij} \log\, \pi_{jk} + (1 - x_{ij}) \log(1 - \pi_{jk})$$

where n_k is the number of data points in class k, with $\sum_{k=1}^{L} n_k = n$.

By maximising with respect to ϕ_k and π_{jk} we obtain the maximum likelihood estimates [64]:

$$\hat{\phi}_k = \frac{n_k}{n} \qquad \hat{\pi}_{jk} = \frac{n_{jk}}{n_k}$$

where $n_{jk} = \sum_{i:y_i=k} x_{ij}$. Fitting a Bernoulli Naïve Bayes model then is swift, since it only requires computing the sums of binary variables.

Laplace Smoothing

Returning to the estimates for the conditional class probabilities, $\hat{\pi}_{jk} = \frac{n_{jk}}{n_k}$, suppose for a particular training set we observe $n_{jk} = 0$, i.e. there are no occurrences of feature j within class k. Our model will set $\hat{\pi}_{jk} = 0$ despite the fact that the true probability might be non-zero and the absence of observations is simply due to undersampling. A method recommended to prevent the occurrence of zero probability estimates is Laplace smoothing [65], which can be thought of as adding pseudocounts to our data.

Our estimate $\hat{\pi}_{jk}$ would then become

$$\hat{\pi}_{jk} = \frac{n_{jk} + \lambda}{n_k + 2\lambda},$$

where $\lambda > 0$ is a smoothing parameter. In this way, values of 0 and 1 are not possible.

2.3.2 SLEEC

Sparse Local Embeddings for Extreme mutli-label Classification (SLEEC) is a state-of-the-art embeddings-based approach to XML whose embeddings preserve distances between points locally, rather than globally, as with most embeddings methods [37]. As the paper notes, existing embeddings approaches which try to learn a low-dimensional linear subspace for the full label space $\mathcal{Y}$ perform poorly in the presence of data with a large number of tail labels, which are particularly common in the NextMove and SureChEMBL data sets[1]. However they *can* be accurately modelled using a low-dimensional non-linear manifold.

Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{n}$ be a training set such that $\mathbf{x}_i \in \mathcal{X} \subseteq \mathbb{R}^d$ are the data vectors and $\mathbf{y}_i \in \mathcal{Y} \subseteq \{0,1\}^L$ the label vectors where y_{ij} has a 1 if and only if the j^{th} label is present for point i. We then let $\mathrm{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ and $\mathrm{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$ be the data and label matrices. Given $\mathcal{D}$, the goal then is to learn a multi-label classifier $f : \mathbb{R}^d \to \{0,1\}^L$ which can accuractely predict the labels of a new point.

The goal of any embedding approach is to reduce the high dimensionality of the label space (take for example the SureChEMBL data set which contains over a million patents) and embed it in a smaller space of dimension $\hat{L}$ (usually several hundred) in order to train the model. SLEEC works by mapping label vectors $\mathbf{y}_i \in \{0,1\}^d$ to $\mathbf{z}_i \in \mathbb{R}^{\hat{L}}$ and then learning a set of reggressors $\mathrm{V} \in \mathbb{R}^{\hat{L} \times d}$ such that $\mathbf{z}_i \approx \mathrm{V}\mathbf{x}_i$.

Embedding

The embedding matrix $\mathrm{Z} \in \mathbb{R}^{\hat{L} \times n}$ is computed by minimising the objective function:

$$\min_{\mathrm{Z} \in \mathbb{R}^{\hat{L} \times n}} \|P_\Omega(\mathrm{Y}^\top \mathrm{Y}) - P_\Omega(\mathrm{Z}^\top \mathrm{Z})\|_F^2 + \lambda \|\mathrm{Z}\|_1.$$

Here Ω is the set of indices for which the nearest neighbours are preserved, i.e. $(i,j) \in \Omega$ if and only if $j \in \mathcal{N}_i$ where $\mathcal{N}_i$ is the set of nearest neighbours of i. $P_\Omega : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$ is defined:

$$(P_\Omega(\mathrm{Y}^\top \mathrm{Y}))_{ij} = \begin{cases} \langle \mathbf{y}_i, \mathbf{y}_j \rangle & \text{if } (i,j) \in \Omega \\ 0 & \text{otherwise,} \end{cases}$$

where $\langle \cdot, \cdot \rangle$ is the dot product. The mapping P_Ω can be thought of as “sparsifying” the inner product matrix $\mathrm{Y}^\top \mathrm{Y}$, where the $(i,j)^{\text{th}}$ entry is set to zero if $(i,j) \notin \Omega$ and leaves other elements as they are.

The L1 regularisation term, $\|\mathrm{Z}\|_1 = \sum_i \|\mathbf{z}_i\|_1$, is also added to the objective function in order to promote a sparse embedding. A sparse embedding has the benefit of using less memory, reducing prediction time, and avoiding overfitting [37].

[1]See Section 1.7.3 and Figure 1.4 for more detail.

Now given the embedding $\mathrm{Z} = [\mathbf{z}_1, \ldots, \mathbf{z}_n] \in \mathbb{R}^{\hat{L} \times n}$, we wish to learn a regression model that can predict Z given the input features. That is, we wish to learn $\mathrm{V} \in \mathbb{R}^{\hat{L} \times d}$ such that $\mathrm{Z} \approx \mathrm{VX}$.

Combining the two formulations and adding L2 regularisation for V we arrive at our second objective function:

$$\min_{\mathrm{V} \in \mathbb{R}^{\hat{L} \times d}} \|P_\Omega(\mathrm{Y}^\top \mathrm{Y}) - P_\Omega(\mathrm{X}^\top \mathrm{V}^\top \mathrm{VX})\|_F^2 + \lambda \|\mathrm{V}\|_F^2 + \mu \|\mathrm{VX}\|_1.$$

With V known, we can predict the labels for a new point $\mathbf{x}$ by computing $\mathbf{z} = \mathrm{V}\mathbf{x}$ and finding the KNN of the set $[\mathrm{V}\mathbf{x}_1, \ldots, \mathrm{V}\mathbf{x}_n]$.

Clustering

In order to scale to large data sets, the training data is clustered using K-Means into C partitions before the embedding takes place. Then for each cluster we learn a set of embeddings before computing regression parameters V^τ for $0 \leq \tau \leq C$. For each test point $\mathbf{x}$, we first determine the closest cluster τ, then find the embedding $\mathbf{z} = \mathrm{V}^\tau \mathbf{x}$ before finally predicting the label vector using KNN in the embedded space.

In order to prevent instabilities from clustering high dimensional data, the SLEEC method uses an ensemble of learners where each base learner is trained using different cluster partitions. This is achieved by using different initialisation points in the clustering procedure to generate different partitions for each learner.

A high level summary of the process of training a SLEEC model and predicting a new point then is:

Algorithm 1 Train SLEEC

```
Require: Data: $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$. Embedding dimensionality: $\hat{L}$. Number of
  clusters: $C$. Number of embedding neighbours: $\bar{n}$.

  Cluster X into partitions $Q^1, \ldots, Q^C$
  for each partition $Q^j$ do
    Form $\Omega$ using the $\bar{n}$ nearest neighbours of each label vector $\mathbf{y}_i$ in $Q^j$
    Compute the embedding $\mathrm{Z}^j$ using $\Omega$
    Learn the regressors $\mathrm{V}^j$ such that $\mathrm{Z}^j \approx \mathrm{V}^j \mathrm{X}^j$
  end for
  return $\{(Q^j, \mathrm{Z}^j, \mathrm{V}^j)\}_{j=1}^C$
```

Algorithm 2 Predict a test point

```
Require: Embeddings: $\{Z^j\}_{j=1}^C$. Reggressors: $\{V^j\}_{j=1}^C$. Partitions: $\{Q^j\}_{j=1}^C$. Test
    point: $\mathbf{x}$. Number of nearest neighbours: $\bar{n}$. Number of desired labels: $p$.

    Determine $Q^\tau$, the partition closest to $\mathbf{x}$
    $\mathbf{z} = V^\tau \mathbf{x}$
    $\mathcal{N}_z = \bar{n}$ nearest neighbours of $\mathbf{z}$ in $Z^\tau$
    $P_x =$ the empirical label distribution for points in $\mathcal{N}_z$
    $y_{\text{pred}} = \text{Top}_p(P_x)$
```

SLEEC Python Implementation

I used a Python implementation of the SLEEC algorithm [66] that differs from the original algorithm described in the paper in a number of ways:

1. The paper uses the dot product similarity measure in order to build the kNN graph of the label vectors. This is not supported in `scikit-learn` and so the implementation uses cosine similarity.
2. Instead of the Singular Value Projection (SVP) [45] used by the paper to learn the low rank embedding, the implementation uses Alternating Least Squares (ALS) [67]. The primary difference between the two methods is that SVP utilises an L1 regularisation term to promote sparsity, whereas ALS uses L2 regularisation.
3. Rather than using Alternating Direction Method of Multipliers (ADMM) [68] with L1 regularisation, the implementation instead uses Lasso regression with L1 regularisation.

Despite these differences, I was able to achieve a very similar classification accuracy of 0.6215 on the benchmark BibTeX data set [69], compared to 0.6532 in the paper.

2.3.3 Model Stacking

Model stacking is a model ensemble approach in which predictions of base learners are used as features to train a second-level meta model, which then outputs its own final prediction. Model stacking has been shown both to improve prediction accuracy as well as reduce the variance of predictions [70].

Dietterich [71] cites three fundamental reasons for building model ensembles:

1. **Statistical**: There is only a small amount of data.
2. **Computational**: There is enough data but local searches only produce local optima.
3. **Representational**: When the true function f cannot be represented by any models in the hypothesis space $\mathcal{H}$.

The representational case is likely to be most relevant for this project.

A learning problem can be viewed as searching the space of hypotheses $\mathcal{H}$ to find the hypothesis h that best approximates the true underlying function f, which predicts the outputs. But what if none of the hypotheses in $\mathcal{H}$ can accurately represent f? The situation is depicted in Figure 2.5. Since the data sets considered in this project

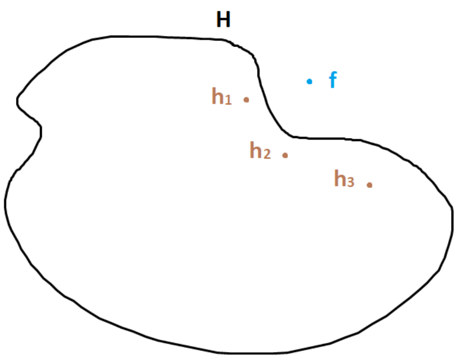


Figure 2.5: Illustration of a hypothesis space of possible models, $\mathcal{H}$. Sometimes $\mathcal{H}$ does not contain the true underlying function, f.

are very large, $\mathcal{H}$ is constrained by memory limits imposed by hardware. However, combining predictions from base models does not require storing those models in memory, only their final predictions. Hence by stacking base model predictions, we are effectively expanding $\mathcal{H}$ beyond what would be otherwise be possible given the constraints on memory.

Model stacking requires training two layers of models and so three data sets will be needed; train, test, and validation. Since a number of patents contain only two listed compounds, these must be excluded from the analysis if a model stacking approach is used.

Two simple types of model stacker are considered, Logistic Regression and majority voting. Simple models are chosen here so that additional model hyperparameters, which might require tuning, are not added to the overall ensemble. Despite its simplicity, Logistic Regression has been shown to perform well as a model stacker [72]. Similarly, majority voting is the basis for many state-of-the-art ensemble classifiers such as Random Forest [73] and Extremely Randomised Trees [74] discussed below.

2.3.4 Model Ensemble Base Classifiers

In selecting base learners for a model ensemble, we must ensure that they can operate fully on sparse matrices (models which explicitly convert the data to a dense matrix format will immediately overload the memory) and that they can deal adequately with the large feature space.

Random Forest

A Random Forest is an ensemble of decision trees in which each base learner is trained on a bootstrapped sample of the data with node partitioning using only a randomly selected subset of the features [73].

Decision trees are conceptually simple and readily interpretable and can be used for both regression and classification tasks. They are hierarchically structured and work by partitioning the data based on values of particular features [75].

Decision trees have many desirable practical properties [76]:

- They are non-parametric and so do not rely on any assumptions *a priori.*
- They deal well with heterogeneous data (both ordered and categorical).
- They are robust in the presence of outliers and errors in labels.
- Their simplicity means they are easily interpreted.

Figure 2.6 provides a basic example.

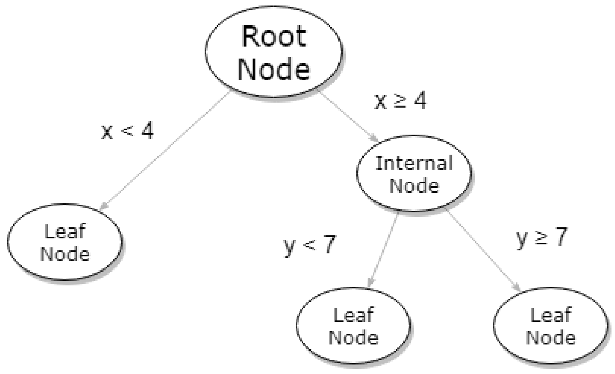


Figure 2.6: Example of a decision tree.

Decision trees are comprised of nodes connected by branches. The root node has no inbound branches while leaf nodes have no outbound branches. Internal nodes contain both inbound and outbound branches. Each internal node partitions the data according to a function of the features (e.g. $x < 4$ vs $x \geq 4$ in Figure 2.6) in which the splitting criteria must be mutually exclusive in order to prevent a data point being mapped to two different leaf nodes.

Optimal values for the threshold used at a node are determined using a partitioning criterion based on the *impurity* of the node, the most common of which are the GINI impurity and the entropy [75]. For a random variable X which takes values in the set $\{a_k\}_{k=1}^{K}$, the GINI impurity, $G(X)$, is defined as:

$$\mathrm{G}(X) = \sum_{k=1}^{K} \mathrm{P}(X = a_k)(1 - \mathrm{P}(X = a_k))$$

and the entropy, $H(X)$, is defined as:

$$\mathrm{H}(X) = -\sum_{k=1}^{K} \mathrm{P}(X = a_k) \times \log_2 \mathrm{P}(X = a_k).$$

The idea is that purer nodes will more appropriately partition the data and lead to better predictions. Starting with all data in a single (root) node, we greedily grow a decision tree by iteratively splitting nodes into purer subnodes until all leaf nodes cannot be made any purer.

Allowing decision trees to grow until all leaf nodes have maximum purity can lead to overfitting [76] and so to overcome this it may be necessary to introduce stopping criteria. Examples of stopping criteria include setting a maximum tree depth (where tree depth is defined as the size of the longest path from root to leaf, referred to as `max_depth` in `scikit-learn`), by only allowing splits to occur on nodes which contain a minimum number of samples (`min_samples_split` in `scikit-learn`), or by requiring each leaf node to contain a minimum number of samples for a split to be valid (`min_samples_leaf` in `scikit-learn`).

Another method used to overcome overfitting of the training data is an ensemble of decision trees, where random perturbations are introduced to the learning procedure in order to produce several different models trained on the same data.

One example of building an ensemble of decision trees is bagging [77], which takes its name from **b**ootstrapping and **agg**regat**ing** individual learners. Bootstrapping is the process of repeated sampling of the original training data with replacement in order to generate new simulated data sets [78]. A seperate model is trained for each bootstrapped data set and then the average of the predictions is taken as our final bagged model.

Given initial training data X, B bootstrapped data sets $\{X^{(b)}\}_{b=1}^{B}$ are generated by sampling with replacement and then separate learners $\hat{f}^1, \ldots, \hat{f}^B$ are trained on each. We then combine them into a final learner and can predict a new point x by

$$\hat{f}_{\text{Bagged}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^b(x).$$

In a classification context, base learners can be thought of as members of a committee where the final class prediction, $\hat{y}_{\text{Bagged}}$, is the majority vote of the base models. If $\hat{y}^1, \ldots, \hat{y}^B$ are the class predictions of the base learners, then

$$\hat{y}_{\text{Bagged}} = \arg\max_{k \in y} \sum_{b=1}^{B} \mathbb{I}(\hat{y}^b = k).$$

A Random Forest is similar to a bagged ensemble but with the key difference that at each splitting decision, Random Forest only considers a randomly chosen subset of all features [73]. Combining randomness both from the random variable selection as well as the bootstrapped training samples has led to Random Forests often performing at state-of-the-art levels for classification tasks [79].

Random Forests are not often trained on such high dimensional data ($p = 479,611$ features for a single partition of the NextMove set on which the final model will be trained). For example, in a paper [80] comparing standard Random Forests with forests comprised of “oblique” decision trees explicitly designed to handle high dimensional data, the maximum number of features used was still only 15,154.

A possible explanation as to how a Random Forest can adequately deal with such high dimensional data might be due to its sparsity. When computing the partition of a node, the default number of features to consider is $\sqrt{p} = \sqrt{479,611} = 693$. Since the average number of substructures (and hence the average number of non-zero features) for a molecule in the data is 52.37, the probability of observing a 0 for all of the 623 features (randomly sampled without replacement) for the average molecule would be

$$p(693 \text{ features} = 0) = \prod_{i=0}^{692}\Big(1 - \frac{52.37}{479,611 - i}\Big) = 0.934.$$

Hence when determining the split for a particular node, most points in the node will have all zero entries for the 693 features considered. This might result in computationally efficient partitioning and could be an explanation as to how the algorithm runs so quickly despite such a large number of features. However, it should be stressed that this is only conjecture at this point and has not been rigorously validated.

Extremely Randomised Trees

Extremely Randomised Trees is another popular ensemble of random decision trees similar to a Random Forest, but with the key difference being that *all* of the data is used to train each learner (as opposed to a bootstrapped sample) and additional randomness is injected via the splitting procedure [74].

When splitting a node, as with Random Forests, we randomly choose $\hat{p}$ features present in the sample on which to consider splitting. For the j^{th} feature, we *randomly* generate a cut point a_c by sampling uniformly $a_c \sim \mathrm{U}(a_{\min}, a_{\max})$ where $a_{\min}$ is the minimum value of feature j in the sample and $a_{\max}$ is its maximum value in the sample. We then split on the feature which results in the greatest increase in purity.

Due to its simple node splitting procedure, Extremely Randomised Trees is often more computationally efficient than other random decision tree ensembles [74].

AdaBoost

Boosting refers to the process of constructing a “strong” learner by iteratively combining a collection of “weak” learners, each of which may only perform slightly better than random chance [81]. AdaBoost is one such boosting algorithm, which *adaptively* adds weak learners which perform better on sections of the data on which the previous ensemble performed poorly [82].

Given data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$, we initially assign each training point an equal weight, $\omega_i = \frac{1}{n}$. We then train a weak learner $h^{(1)}$ on our data, record which points were misclassified, and assign greater weights to these misclassified points. Then iterating for $t = 2, \ldots, T$, we train a new weak learner $h^{(t)}$, by minimising the weighted classification error

$$\epsilon^{(t)} = \sum_{i=1}^{n} \omega_i^{(t)} \mathbb{I}(y_i \neq h^{(t)}(x_i)).$$

In this way subsequent learners can correct for the mistakes of previous iterations of the ensemble.

The weak learners used to construct an Adaboost classifier are not specified by the algorithm and so any classifiers can potentially be used, though decision stumps (decision trees with no internal nodes, just a root and leaf nodes) are often used in practice due to their low bias [82].

XGBoost

XGBoost stands for extreme gradient boosting and implements a gradient boosting decision tree algorithm [83].

Given data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{n}$ and a loss function l measuring the difference between a prediction $\hat{y}_i$ and target y_i, gradient boosting is a process of iteratively adding learners so that the t^{th} iteration involves finding a learner $f^{(t)}$ which minimises the objective:

$$\mathcal{L}^{(t)}(f^{(t)}) = \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f^{(t)}(x_i)) + \Omega(f^{(t)})$$

where Ω is a regularisation term. Gradient boosting takes its name from the fact that it uses a gradient descent algorithm to minimise the loss function at each iteration.

2.4 Bayesian Optimisation

Given the dimensionality of the hyperparameter space of our stacked models finding the optimal combination of parameters is difficult, and a grid search method would not be computationally feasible. I therefore used a Bayesian Optimisation (BO) of the hyperparameters in order to search for the global minimum. BO is an optimisation approach which attempts to find the global optimum of a black-box function where the internal structure is unknown, and hence it is unknown whether the function is convex, multi-modal, or contains random noise [84].

Moreover, function evaluations may be extremely expensive (such as in the case of training a large machine learning model and predicting a test set, or even carrying out individual drug trials or some other physical experiment) and so BO is specifically designed to minimise the required function evaluations to find a global minimum [85].

We wish to minimise (or maximise, by minimising $-f$) the function $f : \mathcal{X} \to \mathbb{R}$ over a bounded domain $\mathcal{X} \subseteq \mathbb{R}^d$, i.e. we wish to solve the equation:

$$x_{\min} = \mathrm{argmin}_{x \in \mathcal{X}} f(x).$$

Since f is a black-box function, we are only able to observe pointwise evaluations $y_i = f(x_i) + \epsilon_i$, where we assume $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. The BO approach uses a surrogate model of the black-box function f in order to perform the optimisation. The most common approach to modelling f is to use a Gaussian Process.

Gaussian Processes

A Gaussian Process (GP) is a collection of random variables, any finite subset of which are jointly normally distributed [86]. If we have an index set $\mathcal{X}$, a collection of random variables $\{A_x\}_{x \in \mathcal{X}}$ is a GP if and only if for every finite set of indices $[x_1, \ldots, x_n]$ the vector $[A_{x_1}, \ldots, A_{x_n}]$ is distributed as a multivariate normal distribution on $\mathbb{R}^n$ [87].

Thus we can consider a GP as a distribution over functions, and so we can model our black-box function f as being drawn from some GP,

$$f \sim \mathcal{GP}(m, k),$$

where

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)] \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \end{aligned}$$

are the mean and covariance functions respectively [87]. Now suppose we have a set of inputs $\mathbf{x} = \{x_i\}_{i=1}^n$ on which we wish to evaluate our objective function f resulting in the vector $\mathbf{f} = [f(x_1), \ldots, f(x_n)]^\top \in \mathbb{R}^n$ and we further have a vector of observed outputs $\mathbf{y} = [y_1, \ldots, y_n]^\top \in \mathbb{R}^n$. Recall that our outputs $y_i = f(x_i) + \epsilon_i$ are noisy evaluations of the underlying function f. We then have prior and likelihood functions[2]

$$\begin{aligned} \mathbf{f} &\sim \mathcal{N}(0, \mathbf{K}) \\ \mathbf{y}|\mathbf{f} &\sim \mathcal{N}(\mathbf{f}, \sigma^2 I), \end{aligned}$$

[2]Note here that we have assumed a zero mean prior for our function f. If we have reason to believe that the true mean of the function is non-zero *a priori* we could simply use the centred function $f'(x) = f(x) - m(x)$ and proceed with the formulation above.

where $\mathbf{K}$ is the covariance or kernel matrix given by $\mathbf{K}_{ij} = k(x_i, x_j)$, and I is the identity matrix.

We now make use of a result from Gaussian conditioning [88]: let $\mathbf{z} \sim \mathcal{N}(\mu, \Sigma)$ be a multivariate normal distribution which we can decompose as

$$\mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{bmatrix}, \ \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \ \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}.$$

Then the conditional distribution of $\mathbf{z}_2$ given $\mathbf{z}_1$ is normal and given by

$$\mathbf{z}_2 | \mathbf{z}_1 \sim \mathcal{N}(\mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{z}_1 - \mu_1), \ \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}).$$

Using this result we have that $\mathbf{f}$ and $\mathbf{y}$ are jointly normal with

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K} & \mathbf{K} \\ \mathbf{K} & \mathbf{K} + \sigma^2 I \end{bmatrix} \right).$$

Finally we apply the Gaussian conditioning rule above to get the posterior distribution of $\mathbf{f}$ given observed noisy evaluations $\mathbf{y}$:

$$\mathbf{f} | \mathbf{y} \sim \mathcal{N}(\mathbf{K}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{y}, \ \mathbf{K} - \mathbf{K}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{K}).$$

From this we can obtain the *posterior predictive distribution* of the function f applied to a new point x. By again making use of the jointly normal distributions of $\mathbf{f}, \mathbf{y}$, and $f(x)$ we can obtain closed form expressions for the posterior predictive mean and variance[3]. Given previously observed data and evaluations $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, we have

$$f(x) | \mathcal{D} \sim \mathcal{N}(\mu(x), \kappa(x, x)),$$

where

$$\begin{aligned} \mu(x) &= \mathbf{k}_{x\mathbf{x}}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{y}, \\ \kappa(x, x) &= k(x, x) - \mathbf{k}_{x\mathbf{x}}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{k}_{\mathbf{x}x}. \end{aligned}$$

Here $\mathbf{k}_{x\mathbf{x}}$ is a $1 \times n$ vector whose j^{th} element is $k(x, x_j)$, $\mathbf{k}_{\mathbf{x}x} = \mathbf{k}_{x\mathbf{x}}^\top$, and $k(x, x)$ is the kernel function of the GP prior for f[4]. The posterior predictive mean and variance will guide the next choice of evaluation point by using an *acquisition function*. A good acquisition function will balance exploitation and exploration, i.e. exploiting areas with low posterior mean $\mu(x)$ and exploring areas with large posterior variances $\kappa(x, x)$.

[3]See Appendix A.1 for further detail.

[4]That is, $k(x, x') = \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]$, where $m(x) = \mathbb{E}[f(x)]$

There are many choices of acquisition function but one with widespread use is the Expected Improvement. First, we define the utility:

$$u(x) = \max(0, \tilde{y} - f(x)),$$

which measures the improvement from $\tilde{y}$, the current minimum. We then wish to maximise $\mathbb{E}[u(x)|\mathcal{D}]$ for the set of previously observed function evaluations $\mathcal{D}$.

A highly desirable property of BO is that the uncertainty of the distribution over the space of functions is readily quantifiable and credible intervals for the posterior mean at a given point can be easily calculated.

Bayesian Optimisation Example

Below we provide a toy illustrative example using BO to determine the maximum of the function $f(x) = \sin(x)$ over the domain $\mathcal{X} = [-\pi, \pi]$.

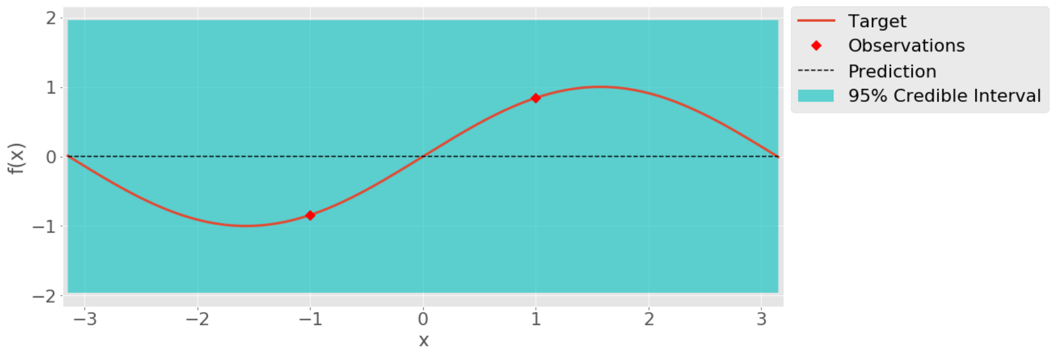


Figure 2.7: Initialisation of the Bayesian Optimisation.

We can see in Figure 2.7 that we use an uninformative prior for our function f and then initialise the optimisation by evaluating the points $x = \{-1, 1\}$. We then update our beliefs after evaluating these points and show the new posterior distribution in Figure 2.8.

The dotted line represents the posterior mean for the value of f at a given value of x in the domain, while the region shaded in blue encloses a 95% credible interval for the value of f. We plot the acquisition function below the main figure which shows the expected improvement of the maximum for the next evaluation of the function.

Figure 2.9 shows the result of a further function evaluation and update of the posterior distribution and acquisition function. We can see the acquisition function

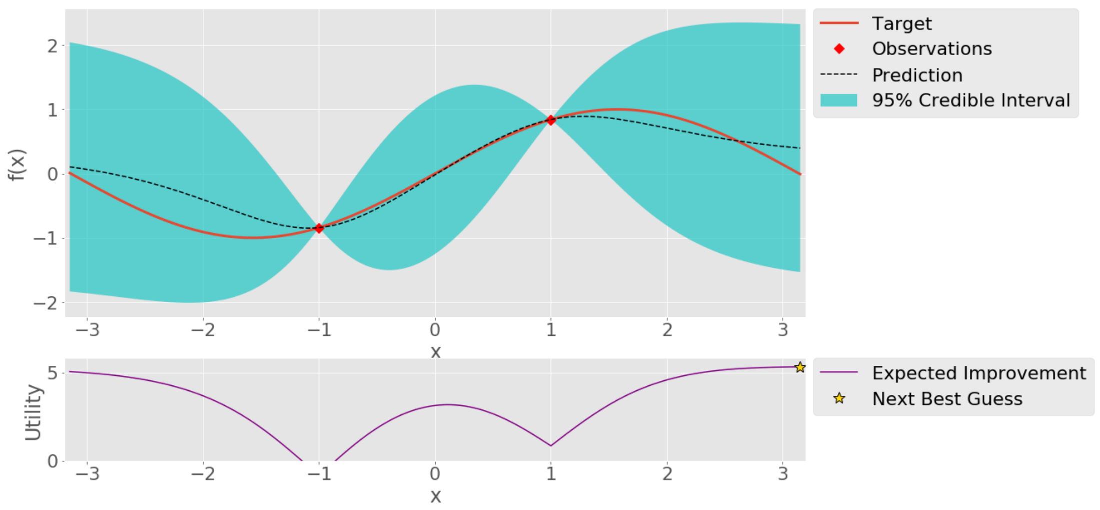


Figure 2.8: Bayesian Optimisation after updating the posterior distribution of f.

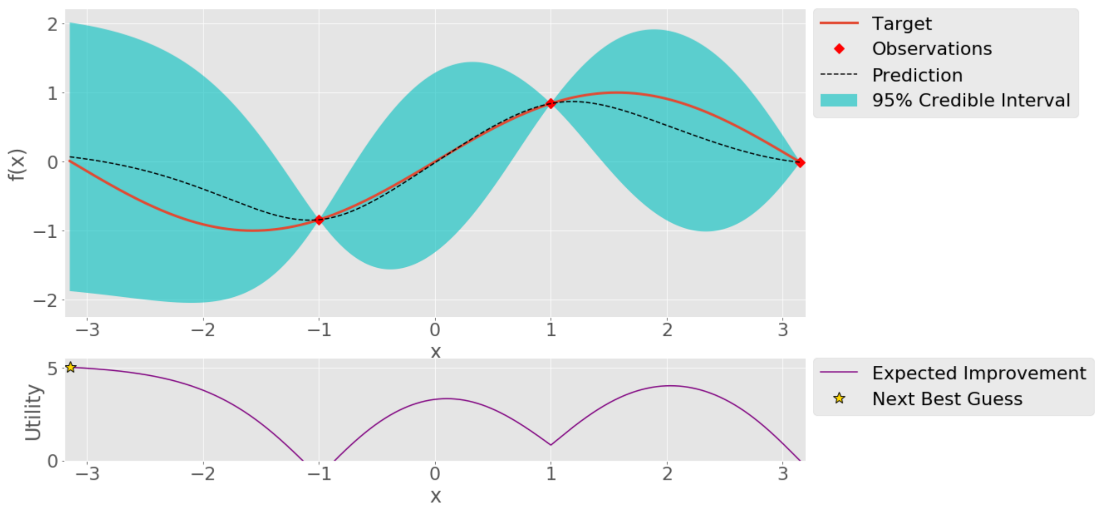


Figure 2.9: Bayesian Optimisation after a further iteration.

here prioritising exploration of unknown portions of the space when it suggests evaluating the point $x = -\pi$ despite the posterior mean being greatest in the region around $x = 1.5$.

Finally, after several more evaluations, in Figure 2.10 we observe the algorithm prioritising exploitation; even though the uncertainty in the region around $x = -2$ is quite large, it is still much less likely to provide an improvement on the maximum than the areas around $x = 1.5$, even though the uncertainty there is very low.

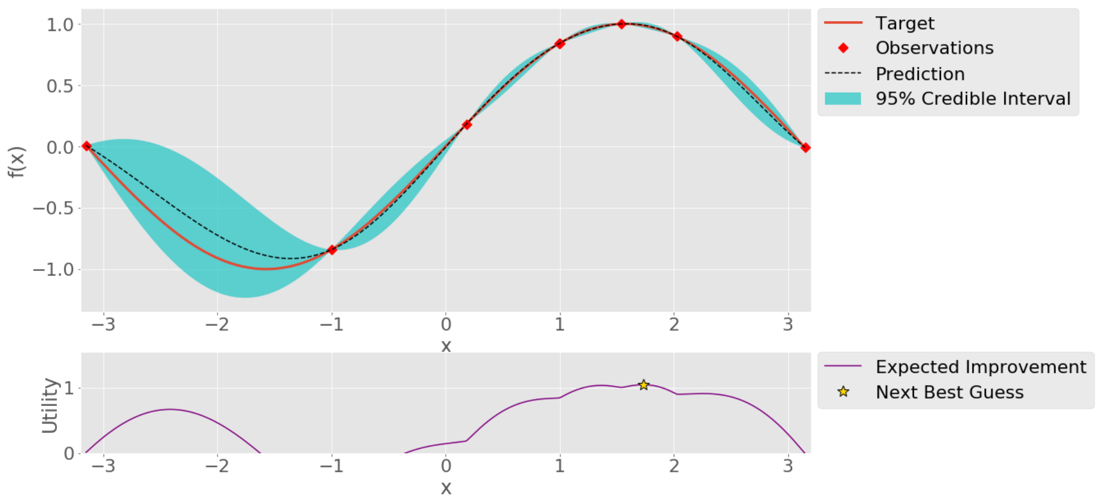


Figure 2.10: Bayesian Optimisation after four more iterations.

A particular benefit of Bayesian Optimisation is that the expensive evaluation of functions can be parallelised and processes only need to be synchronised in order to update the posterior distribution and determine the next batch of points to evaluate. BO has been shown to outperform expert machine learning practitioners in the tuning of model hyperparameters [89].

Ce qui est simple est toujours faux.
Ce qui ne l'est pas est inutilisable.

What is simple is always wrong.
What is not is unusable.

— Paul Valéry [90]

3 Naïve Bayes Approach

Contents

3.1 Introduction

Before attempting to implement novel solutions to the problem, I created and tested an open source model in order to reproduce the existing proprietary method to classify compounds according to the patents under which they are covered. This chapter consists of a discussion of the work carried out by Dr. van Hoorn at Exscientia (co-supervising this project) in fitting a Naïve Bayes model to a subset of the SureChEMBL data after which I will describe my implementation.

3.2 Previous Work

Despite widespread use of machine learning models applied within chemoinformatic domains, little work has been done in attempting to classify compounds according to their similarity to existing patents.

As a starting point, Dr. van Hoorn has created a proof-of-concept Naïve Bayes model using Pipeline Pilot and has achieved encouraging results [7]. Pipeline Pilot is a proprietary software package used to process and analyse data, primarily tailored for users in the natural sciences [91].

Dr. van Hoorn's model makes use of one-vs-all classification techniques, where a separate binary learner is trained for each patent, which then outputs a confidence of a new molecule belonging to a patent. The patent with the largest confidence value is chosen as the output of the combined model. His model is based on the Naïve Bayes method described in [92].

The original model was trained on a subset of 1,000 patents containing 15,500 compounds taken from the SureChEMBL set and achieved a classification accuracy of 60.1%.

A particular benefit of this model is its interpretability; since probability predictions are provided for all possible substructures seen throughout the data set, we can compare substructures within a molecule to examine the relative contributions to their belonging to a patent.

Figure 3.1 shows an example molecule which was predicted as confidently belonging to a particular patent. Substructures are shaded according to the weight ascribed by the model towards their contributing to the prediction (with darker shades implying a higher confidence level), allowing for quick visual interpretation of the similarity of the substructures to those claimed in the patent.

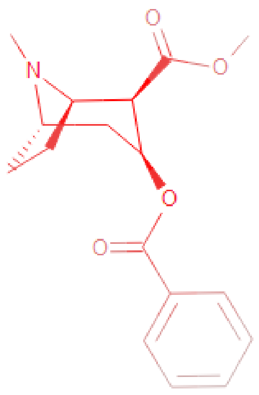


Figure 3.1: The shading of the substructures above corresponds to the degree to which the model believes they contribute to the patent. Reproduced with permission from [7].

One drawback of the one-vs-all approach is that it compares predicted probabilities across multiple Naïve Bayes models. While Naïve Bayes performs competitively for classification tasks [62], the naïve assumptions inherent in the model mean that the posterior probabilities produced often perform poorly [93]. Additionally, because each patent is examined separately in a one-vs-all paradigm, the model does not make use of correlations between patents.

3.3 Python Implementation

I now implement the model described above in Python.

The data used is a subset of 1,509 patents of the SureChEMBL data, which consists of 15,597 compounds. 80% of the compounds in each patent are assigned to the training set and the remaining 20% to the test set. Since there are often patents with a small number of molecules, this 80:20 split is not exact, and the resulting training set contains 77.2% of the data with the remaining 22.8% forming test set.

I used the Python package `RDKit` [54] in order to convert the SMILES strings to chemical structures and then to generate ECFP_6 circular fingerprints. From these fingerprints, I extracted the unique identifiers of all substructures of diameter less than or equal to six and used these as the features of the model.

The model is stored in a dictionary of two layers. Dictionaries are Python data structures which contain values indexed by keys. Rather than in an array where each value is ordered and indexed by its position in the array, dictionaries are unordered and so can be accessed swiftly.

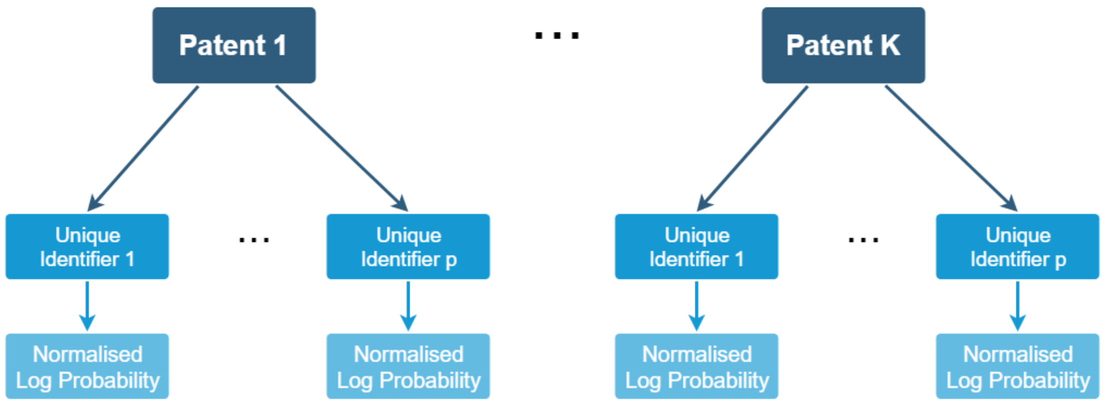

Patent 1
...
Patent K
Unique Identifier 1
...
Unique Identifier p
Unique Identifier 1
...
Unique Identifier p
Normalised Log Probability
Normalised Log Probability
Normalised Log Probability
Normalised Log Probability

Figure 3.2: The form of the model dictionary for data with K patents and p unique identifiers.

The keys of our outer model dictionary correspond to the unique patents, and the value assigned to each patent is itself a dictionary. This inner dictionary contains as *its* keys the unique identifiers of all molecules, and the corresponding values are the normalised log probabilities of the molecule belonging to the patent. The dictionary structure has been visualised in Figure 3.2 above.

3.4 Results

Evaluating the model on the test set, I achieved an accuracy of 16.9%. While quite low, this may be a symptom of the messiness of the data. Alongside molecules and their patents, SureChEMBML also lists the corpus frequency of the molecule in question, defined as the total number of patents in which the molecule appears. If a molecule appears in a large number of different patents, it is unlikely to be the focus of the patent in question and could be simply a reagent listed as part of the production of the compound of interest. This adds unnecessary noise to the model and so I experimented with excluding all compounds with corpus frequencies above a certain limit. The results are shown in Figure 3.3.

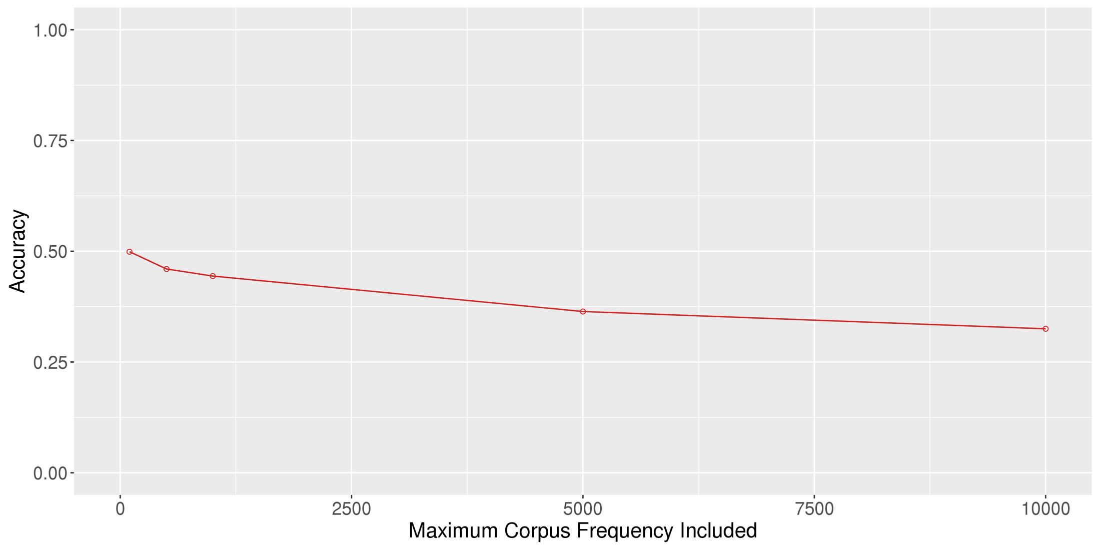


Figure 3.3: Plot of the accuracy of the Naïve Bayes model against maximum corpus frequency.

After excluding common compounds, the predictive power of the model increases significantly, to an accuracy of 49.9% when molecules with a corpus frequency above 100 are excluded. There is a trade-off here between including too many high frequency molecules and possibly adding unnecessary noise to the model versus eliminating too many and losing useful information.

3.5 Conclusion

I successfully created a Python implementation of the Naïve Bayes model developed by Dr. van Hoorn in Pipeline Pilot. The model has been shown to perform adequately on small data sets when noisy molecules are excluded and the proof-of-concept that the problem is one which is amenable to modelling has been validated. I will now focus on attempting to implement a more scalable solution on a larger set of patent data.

I beseech you, in the bowels of Christ, think it possible you may be wrong.

— Oliver Cromwell [94]

4 Novel Approaches

Contents

4.1 Introduction

In this chapter I discuss the computational infrastructure used to carry out analysis, detail the various preprocessing methods applied to the data, and outline the steps taken to select a final model, tune hyperparameters, and output label predictions. I examine the results predicted by both the SLEEC Python implementation and a Label Powerset model ensemble with Random Forest and Extremely Randomised Trees models as base learners stacked using majority voting, as well as comparing them to the Naïve Bayes model described in Chapter 3. Finally, I offer an interpretation of the results.

4.2 Computational Infrastructure

Owing to the scale of the data, it was necessary to carry out the computations required throughout the project in a distributed computing environment. Analysis was carried out using the Google Cloud Platform, a distributed computing environment with pre-built compute instances optimised for popular Python machine learning libraries [95]. With compute instances of up to 64 cores and 416 GB of RAM, using a cloud environment meant that I was able to develop much more complex models than would have been possible on a desktop PC.

4.3 Data

The data used for this section is a single partition of the NextMove patent data described in Section 2.2. I opted for the NextMove set over SureChEMBL due to the inclusion of reaction SMILES data, ensuring that agents and reactants are not considered, which are almost certainly not the intended subjects of the patents.

As before, we have an $n \times p$ data matrix $\mathbf{X}$ and an $n \times L$ label matrix $\mathbf{y}$, with $n = 109,047$ unique compounds, $L = 15,091$ patents, and $p = 479,611$ features. The features here as before correspond to the unique ECFP_6 substructures up to diameter 6 of the molecules in the data.

The data is split in three: a training set, on which base learners are trained; a validation set on which second level model stackers are trained; and a test set on which I evaluated the accuracy of the meta model. In order to ensure that each set contains at least one observation from each patent, I removed any patents which contain two or fewer molecules. A split ratio of 60:20:20 was attempted, but due to the large number of patents with only a small number of molecules[1], an exact split was not possible and the result is 47% of the data used for the train set, 25% for the validation set, and 28% in the test set.

[1]See Figure 1.4.

4.4 Model Building and Selection

As has been discussed previously, the two most popular types of model used to approach extreme multi-label classification tasks are embedding-based and tree-based [37]. Unfortunately most state-of-the-art tree-based methods such as FastXML [41] have been implemented using Matlab, and though the code is publicly available, Google Cloud Platform does not support Matlab and so these methods were not available for this project. I therefore compared two types of model; firstly I trained a model using the SLEEC embedding method implemented in Python by Xiao [66], and secondly I used the Label Powerset problem transformation approach [50] combined with an ensemble of tree-based learners. Finally I compared the accuracy of the Naïve Bayes approach with the SLEEC and ensemble methods.

4.4.1 SLEEC

The SLEEC Python implementation was trained on the combined train and validation data sets (since we do not need to train a model stacker for the SLEEC method) and achieved a classification accuracy of 19.61%. This is quite low and is possibly explained by either poor tuning of the seven hyperparameters or an inefficient implementation of the original paper. Unfortunately, due to training times of approximately 19 hours, tuning such a high dimensional hyperparameter space is not possible.

4.4.2 Label Powerset Approach

I now proceed with the Label Powerset problem transformation approach and take each unique combination of patents observed in the data as its own class[2]. Standard multi-class classifiers can then be used to predict the patents to which a compound might belong.

As part of this approach, I built a model ensemble of base learners which were combined via a model stacker and used to output a final class prediction. A model schematic is shown in Figure 4.1.

As base learners, Random Forest, Extremely Randomised Trees, AdaBoost and, XGBoost models were initially trained and combined using either Logistic Regression or majority voting as a model stacker.

[2]See Section 1.6.3.

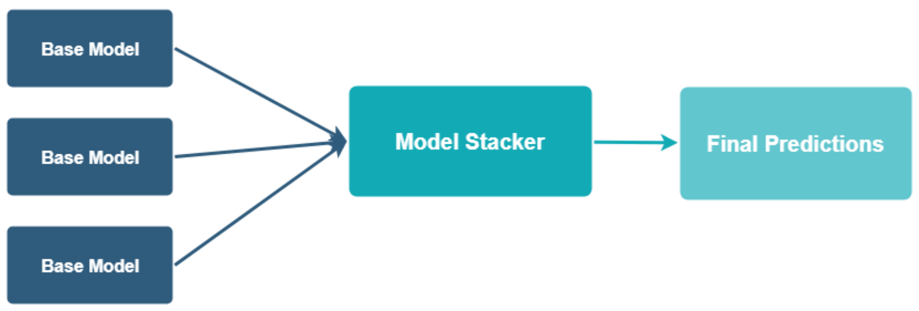


Figure 4.1: Schematic of the model ensemble approach.

Since some base learners require up to 400 GB of memory to run, stacking models in a conventional manner whereby each base model is stored in memory simultaneously is not possible. I therefore instead decided to pre-train the entire space of base learners and store their predictions in the hard drive. These predictions can then be loaded as needed by level two stackers during hyperparameter tuning and final label prediction.

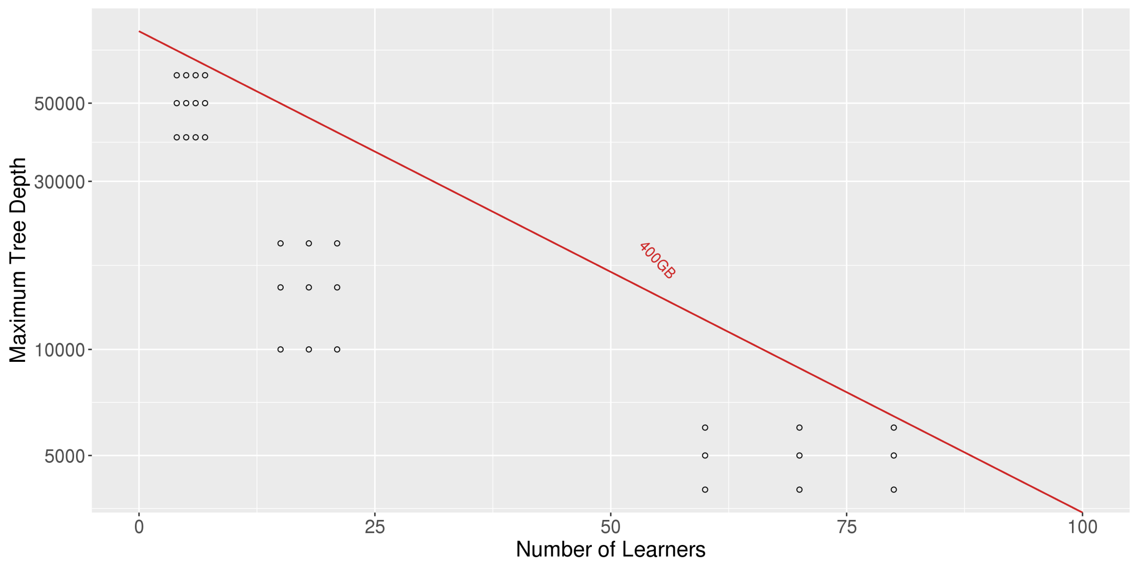


Figure 4.2: Plot of the hyperparameter grid for two Random Forest parameters.

Taking Random Forests as an example, Figure 4.2 shows a two dimensional grid containing the "number of learners" hyperparameter and the "maximum tree depth" hyperparameter. A desirable quality for base models is that they be as different as possible in structure, and so three different "flavours" of model occupying distinct portions of the hyperparameter space above are pre-trained; deep forests in the top left, medium depth forests in the centre, and shallow forests in the bottom right of the plot. For each point on the grid, another hyperparameter is also varied (the minimum number of samples on which to split a node, which takes values in the set $\{2, 4, 6, 8, 10, 12, 14, 16\}$). Each point on the grid therefore represents eight pre-trained models.

The red line in Figure 4.2 is added for illustrative purposes and partitions the hyperparameter space into those models which use less than 400 GB of memory (to the left of the red line) and those which use more than 400 GB of memory (to the right of the red line). While memory usage scales with both the maximum tree depth and the number of learners approximately linearly, the relationship was not exactly determined and the partition above should be thought of as being approximate and only used in order to aid understanding.

I performed similar pre-training for Extremely Randomised Trees models and attempted the same for XGBoost, and AdaBoost base learners but unfortunately training times for both XGBoost and AdaBoost base models took several hours even when the number of learners was low. The performance of these base models was also significantly lower than either Random Forest or Extremely Randomised Trees and so they were omitted as base learners for the stacker.

4.4.3 Hyperparameter Tuning

I used Bayesian optimisation to determine the hyperparameters of the base models to feed to the second level model stackers, the number of base models to include, as well as the type of model stacker to use.

The objective function to maximise is:

$$f(\theta,\ m,\ \beta),$$

where θ corresponds to the hyperparameters of the base models, m is the number of base models to include, and β is a Boolean where $\beta = 0$ stacks the models using Logistic Regression and $\beta = 1$ stacks them using majority voting. The output of the function f is the accuracy of the final stacked model on the test set.

4.5 Results

The optimisation was run in parallel on 16 cores in the cloud in order to expedite compute time using the `GPyOpt` Python package [55]. It was determined that the

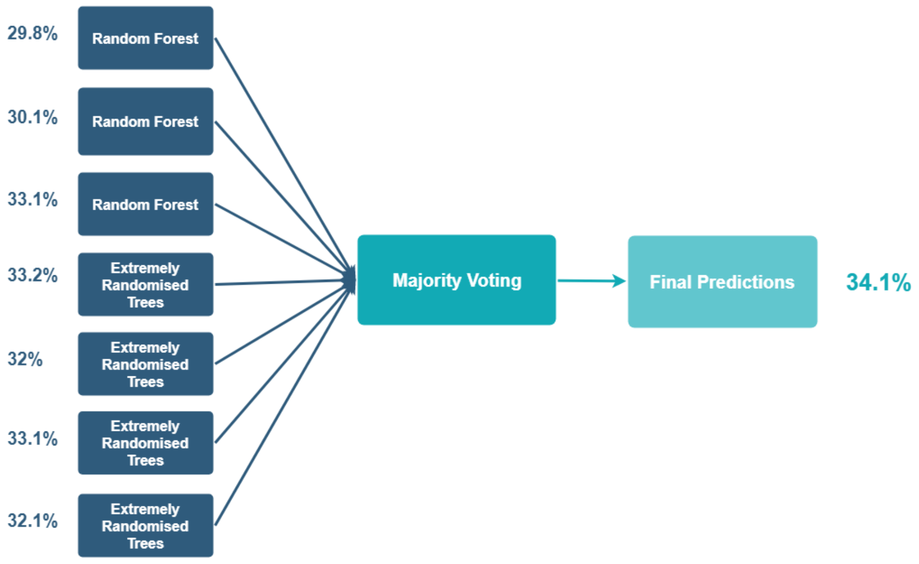


Figure 4.3: Final model schematic.

optimal model consists of three Random Forest and four Extremely Randomised Trees base learners combined using majority voting.

The model ensemble[3] described in Figure 4.3 achieved a classification accuracy of 34.1%, slightly higher than the 33.2% accuracy achieved by a single base learner.

The Logistic Regression stacker did not lead to an increase in performance compared to the top performing base learner. Since training the Logistic Regression requires combining the predicted $n \times L$ label matrices $\hat{\mathbf{y}}$ of each of the base learners, the number of features was mL, where m is the number of base learners used. Since $L = 15,091$ this could lead to tens or even hundreds of thousands of features on which to train the Logistic Regression which may have added too much noise to usefully combine the information. It is also possible that multicolinearity of the features led to a decrease in the performance of the classifier [96].

4.6 Naïve Bayes Comparison

As a final comparison, I compared the SLEEC and model ensemble approaches with the existing Naïve Bayes method described in Chapter 3. Since the Naïve Bayes method would not scale to the data set used in this chapter, models were trained on the SureChEMBL data set described in Section 3.3 with molecules with

[3]See Appendix A.2 for full model hyperparameters.

a corpus frequency greater than 500 removed. The Naïve Bayes method achieved an accuracy of 49.9%, SLEEC achieved a 72.2% accuracy, and the model ensemble method achieved an accuracy of 83.7%.

4.7 Conclusion

Out of the three methods attempted in this project, the Label Powerset and majority voting ensemble performed best by a wide margin. The performance of the SLEEC method, with a classification accuracy of 19.6% was somewhat disapointing but it is difficult to say whether this was a result of improperly chosen hyperparameters or a poorly translated Python implementation of the original Matlab code. In addition, both methods significantly outperformed the Naïve Bayes approach on the smaller data set, despite being optimised for performance on data sets much larger.

Omnium rerum principia parva sunt.

The beginnings of all things are small.

— Marcus Tullius Cicero [97]

5 Conclusions

Contents

5.1 Summary

In this project, I validated existing work in classifying molecules according to their patents by reproducing a proprietary Naïve Bayes model using open source software written in Python before then exploring new methods of tackling the problem. I tested both an existing implementation of the SLEEC algorithm and compared the results to a model ensemble approach using a Label Powerset transformation alongside an ensemble of Random Forest and Extremely Randomised Trees base learners stacked using a majority voting procedure. I then performed hyperparameter tuning of the base models using Bayesian Optimisation and output final model predictions.

The problem I have attempted to solve, building a classification model which encompasses *all* existing molecular patents, was grand in scope. Starting from a simple Naïve Bayes proof-of-concept model trained on 1,000 patents, I have successfully implemented a scalable model pipeline which achieves a reasonable classification accuracy when trained on data from 15,000 patents - just over 10%

of the NextMove database - by reformulating the problem as one of Extreme Multi-Label classification and leveraging the strategies tailored to problems of this kind.

This represents a significant improvement in answering the posed question “Can computers replace patent attorneys?”, and allows for the possibility that a more complete solution may follow. That said, there are some limitations in the analysis which are worth considering.

Firstly, the requirement to split our data into training, validation, and test sets means that any patents which contain fewer than three compounds (25.4% of all patents) must necessarily be excluded from our analysis. This represents a significant proportion of the patent space on which we are unable to test our model. However if the model were ever put into production, it could still be trained using all patents and should not present issues assigning novel compounds to these patents.

Secondly, the MSc dissertation guidelines state that analysis should be carried out on a desktop computer, however it is unlikely to have proven fruitful in our case given the scale of the problem. Successful base models used between 200 and 400 GB of memory and so it is unlikely that models with sufficient complexity could be trained on a desktop PC. Even in a cloud environment with approximately 7,000 core hours available, it took between three and five core hours to train a single base model, and up to ten core hours to train a single second level model stacker. This necessitated careful decisions regarding the application of scarce computational resources.

Finally, though there is a rich literature approaching the problem of Extreme Multi-Label classification, most significant developments have been recent and Python implementations of the original code have so far not been written. Though I used an implementation of the state-of-the-art SLEEC embedding approach, this was not written by the paper authors and has not been rigorously validated.

5.2 Further Work

There are a number of avenues further work on this topic might explore.

Given that the partitioning of the data was done randomly without leveraging relations between the patents, it is possible that stronger performance could be achieved by first clustering the patents to achieve a more efficient partitioning.

The Extreme Classification Repository [69] contains the original Matlab implementations of many of the state-of-the-art extreme multi-label classification models including SLEEC [37] and FastXML [41]. Using these models in a distributed computing environment capable of running Matlab code could yield significant increases in predictive accuracy.

Using a Grammar VAE [24] to map the data to a low dimensional latent space using a VAE specifically tailored for the SMILES grammar could result in a more efficient dimensionality reduction than the TSVD attempted in Section 2.2.4. If an efficient mapping were possible in this way then it might allow for a wider variety of model to be tested.

Finally, a model is only as good as the data used to train it and it is likely that both the SureChEMBL and NextMove sets require significant cleaning. Developing an automated process to clean the noisy data might lead to additional gains in model performance.

Appendices

A
Additional Mathematical Detail

Contents

A.1 Gaussian Process Posterior Predictive Distribution

A Gaussian Process (GP) is a collection of random variables, any finite subset of which are jointly normally distributed [86]. If we have an index set $\mathcal{X}$, a collection of random variables $\{A_x\}_{x \in \mathcal{X}}$ is a GP if and only if for every finite set of indices $[x_1, \ldots, x_n]$ the vector $[A_{x_1}, \ldots, A_{x_n}]$ is a multivariate normal distribution on $\mathbb{R}^n$ [87].

Thus we can consider a GP as a distribution over functions, and so we can model our black-box function f as being a draw from some GP,

$$f \sim \mathcal{GP}(m, k)$$

where

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)] \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \end{aligned}$$

are the mean and covariance functions respectively. Now suppose we have a set of inputs $\mathbf{x} = \{x_i\}_{i=1}^n$ on which we wish to evaluate our objective function f resulting in the vector $\mathbf{f} = [f(x_1), \ldots, f(x_n)]^\top \in \mathbb{R}^n$ and we further have a vector of observed outputs $\mathbf{y} = [y_1, \ldots, y_n]^\top \in \mathbb{R}^n$. Recall that our outputs $y_i = f(x_i) + \epsilon_i$ are noisy evaluations of the underlying function f. We then have prior and likelihood functions

$$\begin{aligned} \mathbf{f} &\sim \mathcal{N}(0, \mathbf{K}) \\ \mathbf{y}|\mathbf{f} &\sim \mathcal{N}(\mathbf{f}, \sigma^2 I), \end{aligned}$$

where $\mathbf{K}$ is the covariance or kernel matrix given by $\mathbf{K}_{ij} = k(x_i, x_j)$.

We now make use of a result from Gaussian conditioning: let $\mathbf{z} \sim \mathcal{N}(\mu, \Sigma)$ be a multivariate normal distribution where we can decompose as

$$\mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{bmatrix}, \ \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \ \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}.$$

Then the conditional distribution of $\mathbf{z}_2$ given $\mathbf{z}_1$ is normal and given by

$$\mathbf{z}_2|\mathbf{z}_1 \sim \mathcal{N}(\mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{z}_1 - \mu_1), \ \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12})[88].$$

Using this result we have that $\mathbf{f}$ and $\mathbf{y}$ are jointly normal with

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{y} \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K} & \mathbf{K} \\ \mathbf{K} & \mathbf{K} + \sigma^2 I \end{bmatrix} \right).$$

Finally we apply the Gaussian conditioning rule above to get the posterior distribution of $\mathbf{f}$ given observed noisy evaluations $\mathbf{y}$:

$$\mathbf{f}|\mathbf{y} \sim \mathcal{N}(\mathbf{K}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{y}, \ \mathbf{K} - \mathbf{K}(\mathbf{K} + \sigma^2 I)^{-1}\mathbf{K}).$$

We can continue in this way to construct the posterior predictive distribution. Given test data $\mathbf{x}' = \{x'_j\}_{j=1}^m$ we extend the model to include the values $\mathbf{f}' = [f(x'_1), \ldots, f(x'_m)]^\top \in \mathbb{R}^m$ of f evaluated at the test points. Note that since our prior was over the full space of functions f, we can extend it to include the evaluations $\mathbf{f}'$. Now we have the model

$$\begin{aligned} \begin{bmatrix} \mathbf{f} \\ \mathbf{f}' \end{bmatrix} |\mathbf{x}, \mathbf{x}' &\sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \mathbf{K}_{\mathbf{xx}} & \mathbf{K}_{\mathbf{xx}'} \\ \mathbf{K}_{\mathbf{x}'\mathbf{x}} & \mathbf{K}_{\mathbf{x}'\mathbf{x}'} \end{bmatrix} \right) \\ \mathbf{y}|\mathbf{f} &\sim \mathcal{N}(\mathbf{f}, \sigma^2 I), \end{aligned}$$

where $(\mathbf{K_{xx}})_{ij} = k(x_i, x_j)$, $(\mathbf{K_{x'x'}})_{ij} = k(x'_i, x'_j)$, $\mathbf{K_{xx'}}$ is an $n \times m$ matrix whose $(i,j)^{\text{th}}$ entry is $k(x_i, x'_j)$ and $\mathbf{K_{x'x}} = \mathbf{K}^\top_{\mathbf{xx'}}$. Now making use of the joint normality of $\mathbf{f'}$ and $\mathbf{y}$:

$$\begin{bmatrix}\mathbf{f'}\\ \mathbf{y}\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}\mathbf{K_{x'x'}} & \mathbf{K_{x'x}}\\ \mathbf{K_{xx'}} & \mathbf{K_{xx}} + \sigma^2 I\end{bmatrix}\right).$$

and from Gaussian conditioning rules again we can read off the posterior predictive distribution

$$\mathbf{f'}|\mathbf{y} \sim \mathcal{N}(\mathbf{K_{x'x}}(\mathbf{K_{xx}} + \sigma^2 I)^{-1}\mathbf{y},\ \mathbf{K_{x'x'}} - \mathbf{K_{x'x}} - (\mathbf{K_{xx}} + \sigma^2 I)^{-1}\mathbf{K_{xx'}}).$$

A.2 Final Model Hyperparameters

The full model hyperparameters of our model are provided in the table below. The model consists of an ensemble of Random Forest (RF) and Extremely Randomised Trees (ERT) base models.

Base Model	Number of Learners	Maximum Depth	Minimum Samples per Split	Minimum Samples per Leaf
RF	60	60,000	6	2
RF	21	10,000	4	1
RF	7	40,000	4	1
ERT	9	40,000	6	1
ERT	25	10,000	4	1
ERT	9	20.000	4	1
ERT	20	10,000	2	1

B

Pruned Code

Contents

B.1 Data Preprocessing

```r

##### Here we load the full data and perform a train:test split.


full_data = read.csv('full_corpus.csv')

# exclude columns we don't need

full_data = full_data[, 2:4]


# change the factor types to character strings for quicker accessing

full_data[, 1] = as.character(full_data[, 1])
full_data[, 2] = as.character(full_data[, 2])

# here we exclude molecules with a corpus frequency greater than 10
# and then partition our data into ten subsets.

full_10 = full_data[full_data[, 3] < 10, ]
n_10 = length(full_10[, 1])

# randomise the rows
full_10 <- full_10[sample(nrow(full_10)),]

for (i in 1:10){
    # this loop writes the ith partition to file in .csv form

    write.csv(full_10[((i-1)*n_10 + 1):floor(i*n_10/10), 1:2], paste('full_10_', i, '.csv', sep
     = ""))
}

##################################################################


```

```r
train_test = function(corpus, subset){

    # function which loads a data set and performs an 80:20 train:test split
    # before writing the two sets to .csv file
    #
    #
    # inputs: corpus - the maximum corpus frequency wanted
    #         subset - which subset of the data you wish to load.
    #                  subset takes values in 1 to 10.
    #
    #
    # outputs: train and test .csv files

    data = read.csv(paste('full_', corp, '_', subset, '.csv', sep = ''))

    sample_data = data[, 2:3]
    sample_data[, 1] = as.character(sample_data[, 1])
    sample_data[, 2] = as.character(sample_data[, 2])

    sample_patents = unique(sample_data[, 2])

    # Now split the data into train and test

    train_indices = c()

    for (i in 1:(length(sample_patents))){

      # This loop samples 80% of the molecules of a particular patent and stores their
      # indices in a vector.

      train_indices = c(train_indices, sample(x = which(sample_data[, 2] == sample_patents[i]),
                                              size = (floor(0.8*length(which(sample_data[, 2]
    == sample_patents[i])))),
                                              replace = FALSE))
    }

    train_indices = sort(train_indices)

    train_data = sample_data[train_indices, ]
    test_data  = sample_data[-train_indices, ]

    # we only keep patents which occur in both the train and test sets

    keep_indices = which(test_data[, 2] %in% train_data[, 2])

    updated_test = test_data[keep_indices, ]

    train_keeps = which(train_data[, 2] %in% updated_test[, 2])
    updated_train = train_data[train_keeps, ]

    # Now we need to test for duplicates

    train_duplicates = which(duplicated(updated_train))
    test_duplicates  = which(duplicated(updated_test))

    # remove duplicates

    train_no_dupes = updated_train[-train_duplicates,]
    test_no_dupes =  updated_test[-test_duplicates,]

    # now write the data to csv

    write.csv(train_no_dupes, paste('train_', corp, '_', subset, '.csv', sep = ''))
    write.csv(test_no_dupes, paste('test_', corp, '_', subset, '.csv', sep = ''))

}

# select a maximum corpus frequency of 10 and the 1st partition
# and call the function

corp = 10
subset = 1

# call the function

train_test(corp = corp, subset = subset)

##################################################################################

library(Matrix)

### Now we will populate the label matrix y


generate_y = function(corp, subset){

    # this function populates the label matrix y

    ## Input: corp = max corp frequency
    #         subset = subset of data

    ## Output: Y_train, Y_test - the label matrices
    ##         train_mols, test_mols - the unique train and test molecules required
    #           for further formatting

    # read the data

    train = read.csv(paste('train_', corp, '_', subset, '.csv', sep = ''))
    test = read.csv(paste('test_', corp, '_', subset, '.csv', sep = ''))

    train = train[, 2:3]
    train[, 1] = as.character(train[, 1])
    train[, 2] = as.character(train[, 2])


    unique_patents = as.character(unique(train[, 2]))
    unique_train = as.character(unique(train[, 1]))

    # generate empty sparse matrix

    train_labels = Matrix(0, nrow = length(unique_compounds), ncol = length(unique_patents))

    for (j in 1:length(unique_train)){

      # this loop populates the label matrix Y_train which
      # will be fed to the main python algorithm.

      compound_ind = which(train[, 1] == unique_train[j])
      ind = rep(0, length(compound_ind))

      for (i in 1:length(ind)){
        ind[i] = which(unique_patents == train[compound_ind, 2][i])
      }
      train_labels[j, ind] = 1

    }

    # now do the same for the test set

    test = test[, 2:3]
    test[, 1] = as.character(test[, 1])
    test[, 2] = as.character(test[, 2])

    unique_test = as.character(unique(test[, 1]))

    test_labels = Matrix(0, nrow = length(unique_test), ncol = length(unique_patents))

    for (j in 1:length(unique_test)){

      # this loop populates the label matrix Y_test which
      # will be fed to the main python algorithm.

      compound_ind = which(test[, 1] == unique_test[j])
      ind = rep(0, length(compound_ind))

      for (i in 1:length(ind)){
        ind[i] = which(unique_patents == test[compound_ind, 2][i])
      }
      test_labels[j, ind] = 1

    }

    # now write the label matrices to file

    writeMM(train_labels, paste('temp_Y_train_', corp, '_', subset, '.csv', sep = ''))
    writeMM(test_labels, paste('temp_Y_test_', corp, '_', subset, '.csv', sep = ''))

    # here we also save the unique train and test molecules.
    # we will need these to generate fingerprints in RDKit

    write.csv(unique_compounds, paste('train_mols_', corp, '_', subset, '.csv', sep = ''))
    write.csv(unique_test, paste('test_mols_', corp, '_', subset, '.csv', sep = ''))
}

corp = 10
subset = 1

# now call the function for the first partition

generate_y(corp = corp, subset = subset)

###########################################################################
```

Listing B.1: Data Cleaning

```

# Here we generate the fingerprints and extract the unique identifiers

import pandas as pd
import numpy as np
#import matplotlib.pyplot as plt

# read the data and name the columns

train_data = pd.read_csv('train_mols.csv')
```

```
test_data  = pd.read_csv('test_mols.csv')

train_data.columns = test_data.columns = ['index', 'smile']

# keep only the smile and patent columns

train = train_data.loc[:, ('smile')]
test  = test_data.loc[:, ('smile')]

print(len(train))
print(len(test))

# import RDKit and generate fingerprints

from rdkit import Chem

from rdkit.Chem import AllChem
from rdkit.Chem import rdMolDescriptors

train_fingerprints = []
test_fingerprints = []


for i in range(len(train)):
    m = Chem.MolFromSmiles(train.iloc[i]) # extract the ith molecule from the training set
    print(i*100/len(train))

    if m != None: # check that it is a valid molecule
        fp = AllChem.GetMorganFingerprint(m, 3) # morgan fingerprint of radius 3
        train_fingerprints.append(list(fp.GetNonzeroElements().keys()))

for i in range(len(test)):
    m = Chem.MolFromSmiles(test.iloc[i])
    print(i*100/len(test))

    if m != None:
        fp = AllChem.GetMorganFingerprint(m, 3)
        test_fingerprints.append(list(fp.GetNonzeroElements().keys()))



# determine patents where the molecule is invalid

train_indices = []
for i in range(len(train)):
    m = Chem.MolFromSmiles(train.iloc[i])
    if m == None:
        train_indices = np.append(train_indices, i)

test_indices = []
for i in range(len(test)):
    m = Chem.MolFromSmiles(test.iloc[i])
    if m == None:
        test_indices = np.append(test_indices, i)

# now save the invalid indices to delete from the label matrices

export_train = np.array(train_indices)
np.save('train_indices.npy', export_train)
export_test = np.array(test_indices)
np.save('test_indices.npy', export_test)

# save the train and test fingerprints to csv

import  csv

with open("train_fingerprints.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(train_fingerprints)

with open("test_fingerprints.csv","w") as f:
    wr = csv.writer(f)
    wr.writerows(test_fingerprints)
```

Listing B.2: Generate Fingerprints

```

##########################################################################

## now we wish to populate the data matrix X

generate_X = function(k, subset){

    ## function to generate the train and test data matrices

    ## Inputs:  train and test fingerprints from RDKit.
    ## Outputs: X_train, X_test - sparse data matrices.

    train = read.csv(paste('train_fingerprints_', k, '_', subset, '.csv', sep = ''),
                            header = TRUE)
    test = read.csv(paste('test_fingerprints_', k, '_', subset, '.csv', sep = ''),
                            header = TRUE)


    ### get the unique substructures using the union function

    un = union(train[, 1], train[, 2])

    for (i in 3:length(train[1, ])){
      un = union(un, train[, i])
    }
    for (i in 1:length(test[1, ])){
      un = union(un, test[, i])
    }

    unique_fingerprints = un

    # initialise an empty train matrix

    X_train = Matrix(0, nrow = length(train[, 1]), ncol = length(unique_fingerprints))

    n_train = length(train[, 1])

    print('starting train loop')

    for (i in 1:n_train){

      # this loop populates the data matrix X_train which
      # will be fed to the main python algorithm.

      ind = rep(0, sum(!is.na(train[i, ])))


      print(paste('train matrix is', i*100/n_train, '% complete'))



      for (i in 1:length(ind)){

        # here X_train(i, j) is set to 1 if molecule i contains
        # unique substructure j

        ind[j] = which(unique_fingerprints == train[i, j])
      }
      X_train[i, ind] = 1
    }

    # now do the test set

    X_test = Matrix(0, nrow = length(test[, 1]), ncol = length(unique_fingerprints))

    n_test = length(test[, 1])

    print('starting test loop')

    for (i in 1:n_test){

      # this loop populates the data matrix X_train which
      # will be fed to the main python algorithm.

      ind = rep(0, sum(!is.na(test[i, ])))

      for (j in 1:length(ind)){
        ind[j] = which(unique_fingerprints == test[i, j])
      }
      X_test[i, ind] = 1

    }

    # save the matrices

    writeMM(X_train, paste('X_train_', k, '_', subset, sep = ''))
    writeMM(X_test, paste('X_test_', k, '_', subset, sep = ''))
}


k = 10
subset = 1

# Now call the function for the 1st data partition with maximum corpus frequency 10

generate_X(k = k, subset = subset)

##################################################################################

# Finally we remove the rows of the label matrix whose compounds could not be read
# by RDKit

library(RcppCNPy)

remove_indices = function(k, j){

    # remove the bad rows from the label matrices

    test_indices  = npyLoad(paste("test_indices_", k, '_', j, ".npy", sep = ''))
    train_indices = npyLoad(paste("train_indices_", k, '_', j, ".npy", sep = ''))

    Y_train = readMM(paste("temp_Y_train_", k, '_', j, sep = ''))
    Y_test  = readMM(paste("temp_Y_test_", k, '_', j, sep = ''))

    updated_train  = Y_train[-train_indices, ]
    updated_test   = Y_test[-test_indices, ]

    writeMM(updated_train, paste('y_train_', k, '_', j, sep = ''))
    writeMM(updated_test, paste('y_test_', k, '_', j, sep = ''))

}

k = 10
j = 1

remove_indices(k = k, j = j)
```

Listing B.3: Generate Data Matrix

```

# First we remove any patents with less than three molecules and then perform a train/test/val
     split.

def delete_rows_csr(mat, indices):
    """
    Remove the rows denoted by ``indices`` form the CSR sparse matrix ``mat``.
    """
    if not isinstance(mat, csr_matrix):
        raise ValueError("works only for CSR format -- use .tocsr() first")
    indices = list(indices)
    mask = np.ones(mat.shape[0], dtype=bool)
    mask[indices] = False
    return mat[mask]

def load_data(k, j):

    # function to load data
    # k = corpus frequency
    # j = subset

    X_train, X_test = mmread('X_train_{}_{}'.format(k, j)).tocsr(), mmread('X_test_{}_{}'.
     format(k, j)).tocsr()
    y_train, y_test = mmread('y_train_{}_{}'.format(k, j)).tocsr(), mmread('y_test_{}_{}'.
     format(k, j)).tocsr()

    return(X_train, X_test, y_train, y_test)


# load the data

X_train, X_test, y_train, y_test = load_data(10, 1)

# now combine the train and test sets

y_full = vstack((y_train, y_test))
X_full = vstack((X_train, X_test))

# now compute the row and column sums of the label matrix y

patent_sums = y_full.sum(axis = 0) # this is the number of molecules per patent
mol_sums = y_full.sum(axis = 1) # the number of patents containing the molecule


# find the indices of the patents containing less than three molecules

ind_0 = np.where(np.array(y_sums) == 0)[1]
ind_1 = np.where(np.array(y_sums) == 1)[1]
ind_2 = np.where(np.array(y_sums) == 2)[1]

ind_all = np.sort(np.append(ind_0, ind_1))
ind_all = np.sort(np.append(ind_all, ind_2))

# remove columns (i.e. patents) with sums less than 3

all_cols = np.arange(y_full.shape[1])
cols_to_keep = np.where(np.logical_not(np.in1d(all_cols, ind_all)))[0]
y_red = y_full[:, cols_to_keep]

# now delete the rows (molecules) which no longer occur in any patents

row_sums = y_red.sum(axis = 1)
row_inds = np.where(np.array(row_sums) == 0)[0]

new_y = delete_rows_csr(y_red, row_inds)
new_X = delete_rows_csr(X_full, row_inds)






```

```
# now we do a train/test/val split
# we want an approximate 60/20/20 split but this will not be exact due to the
# large number of patents with only a small number of molecules.

train_inds = []
test_inds  = []
val_inds   = []

for j in range(new_y.shape[0]):

    # here we split each patent into 60/20/20

    inds = np.where(new_y[:, j].todense() == 1)[0]
    split = np.split(inds, [math.floor(.6 * len(inds)), math.floor(.8 * len(inds))])
    train_inds = np.append(train_inds, split[0]) # first 60% to train
    test_inds = np.append(test_inds, split[1]) # next 20% to test
    val_inds = np.append(val_inds, split[2]) # final 20% to val

# combine the values
train_inds = list(dict.fromkeys(train_inds))
test_inds = list(dict.fromkeys(test_inds))
val_inds = list(dict.fromkeys(val_inds))

# now create the new train, test and val label matrices

new_train_y = new_y[train_inds]
new_test_y  = new_y[test_inds]
new_val_y   = new_y[val_inds]

# and do the same for the data matrices

new_train_X = new_X[train_inds]
new_test_X  = new_X[test_inds]
new_val_X   = new_X[val_inds]


# finally save the train, test, and val sets

mmwrite('train_X', new_train_X), mmwrite('test_X', new_test_X), mmwrite('val_X', new_val_X)
mmwrite('train_y', new_train_y), mmwrite('test_y', new_test_y), mmwrite('val_y', new_val_y)
```

Listing B.4: Train/Test/Validation Split

B.2 Chapter 3: Naïve Bayes

```

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# read the data and name the columns

train_data = pd.read_csv('train.csv')
test_data  = pd.read_csv('test.csv')

train_data.columns = test_data.columns = ['surechem_id', 'smile', 'key', 'corpus_frequency', '
    patent_id',
               'publication_date', 'field', 'field_frequency']

# keep only the smile and patent columns

train = train_data.loc[:, ('smile', 'patent_id')]
test  = test_data.loc[:, ('smile', 'patent_id')]

# import RDKit and generate fingerprints

from rdkit import Chem

from rdkit.Chem import AllChem
from rdkit.Chem import rdMolDescriptors

train_fingerprints = []
test_fingerprints = []
unique_fp = []


for i in range(len(train)):
    m = Chem.MolFromSmiles(train.iloc[i, 0]) # extract the ith molecule from the training set

    if m != None: # check that it is a valid molecule
        fp = AllChem.GetMorganFingerprint(m, 3) # morgan fingerprint of radius 3
        train_fingerprints.append(list(fp.GetNonzeroElements().keys()))
        unique_fp = unique_fp + list(fp.GetNonzeroElements().keys())

for i in range(len(test)):
    m = Chem.MolFromSmiles(test.iloc[i, 0])

    if m != None:
        fp = AllChem.GetMorganFingerprint(m, 3)
        test_fingerprints.append(list(fp.GetNonzeroElements().keys()))
        unique_fp = unique_fp + list(fp.GetNonzeroElements().keys())

unique_fp = list(set(unique_fp))

# create the labels data
train_labels = np.array(train.patent_id)
test_labels = np.array(test.patent_id)
print(len(train_labels))
print(len(train_fingerprints))

# determine patents where the molecule is invalid

train_indices = []
for i in range(len(train)):
    m = Chem.MolFromSmiles(train.iloc[i, 0])
    if m == None:
        train_indices = np.append(train_indices, i)

test_indices = []
for i in range(len(test)):
    m = Chem.MolFromSmiles(test.iloc[i, 0])
    if m == None:
        test_indices = np.append(test_indices, i)


# now delete the labels where the molecule is invalid

temp_train_labels = train_labels
for index in sorted(train_indices, reverse=True):
    temp_train_labels = np.delete(temp_train_labels, index)

temp_test_labels = test_labels
for index in sorted(test_indices, reverse=True):
    temp_test_labels = np.delete(temp_test_labels, index)

print(len(temp_test_labels))
print(len(test_fingerprints))

print(len(temp_train_labels))
print(len(train_fingerprints))


# correct

train_labels = temp_train_labels
test_labels  = temp_test_labels

# compute the frequency of the unique fingerprints in the train set

fp_count = np.zeros(len(unique_fp))

for j in range(len(unique_fp)):
    #print(100*j/len(unique_fp))
    for i in range(len(train_fingerprints)):
        if unique_fp[j] in train_fingerprints[i]:
            fp_count[j] = fp_count[j] + 1

# create a list of unique patents

unique_patents = list(set(train_labels))

# compute the number of molecules within each patent

patent_mols = np.zeros(len(unique_patents))
for i in range(len(unique_patents)):
    print(i*100/len(unique_patents))
    patent_mols[i] = len(np.where(train.iloc[:, 1] == unique_patents[i])[0])


num_mols = len(train.iloc[:,1]) # total number of train molecules

# create a dictionary where the keys are the patents and the values inside are each
# of the features present in the patent.

model = {}

for i in range(len(train_labels)):
    if train_labels[i] in model:
        model.update( {train_labels[i] : model[train_labels[i]] + train_fingerprints[i]} )
    else:
        model.update( {train_labels[i] : train_fingerprints[i]} )


# create the model dictionary

model_dict = {}

for k in range(len(unique_patents)):
    print(k*100/len(unique_patents))

    model_dict.update({ unique_patents[k] : {} }) # create the dictionary entry for the kth
    patent
    Pactive = patent_mols[k]/num_mols              # proportion of 'active' molecules in the
    train set
    K       = 1/Pactive                            # used for the Laplacian correction

    NP = np.zeros(len(unique_fp))                  # NP = normalised probablity of each
    fingerprint in the patent
    for i in range(len(unique_fp)):
        fp = unique_fp[i]
        freq_all = fp_count[i]

        if fp in model[unique_patents[k]]:
            num_good = len(np.where(np.array(model[unique_patents[k]]) == fp)[0])
        else:
            num_good = 0

        Pcorr = (num_good + 1)/(freq_all + K)      # Laplacian corrected probability
        NP[i] = np.log(Pcorr/Pactive)              # normalised probability
        model_dict[unique_patents[k]].update( { unique_fp[i] : NP[i] } ) # update the
    dictionary


def NaiveBayes(test_fingerprints, model_dict, unique_patents):

    # function which evaulates a test molecule and outputs the predicted patent

    # inputs: test_fingerprints - fingerprints of the molecule to be evaluated
    #         model_dict - the model dictionary
    #         unique_patents - the list of model patents

    # output: the index of the most likely patent

    norm_prob = np.zeros(len(unique_patents))


    for k in range(len(unique_patents)):
        # print(k*100/len(unique_patents))
        for i in range(len(test_fingerprints)):
            norm_prob[k] =+ model_dict[unique_patents[k]][test_fingerprints[i]]

    return(np.argmax(norm_prob))


# now compute the predictions

predictions = np.zeros(len(test_fingerprints))
for i in range(len(test_fingerprints)):
    print(i*100/len(test_fingerprints))
    predictions[i] = NaiveBayes(test_fingerprints[i], model_dict, unique_patents)

# check whether the predictions were correct

hit = np.zeros(len(predictions))
for i in range(len(predictions)):
    hit[i] = unique_patents[int(predictions[i])] == test_labels[i]

accuracy = sum(hit)/len(hit)
print(accuracy)
```

Listing B.5: Naive Bayes Model Chapter 3

B.3 Chapter 4: Model Fitting and Tuning

```

import pickle
import pandas as pd
import numpy as np
from numpy.random import seed

from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from skmultilearn.problem_transform import LabelPowerset

from scipy.io import mmread
from scipy.sparse import csr_matrix, vstack, hstack

import matplotlib.pyplot as plt

import time
from scipy import stats

import matplotlib.mlab as mlab

# load the data

```

```python
X_train, X_test, X_val = mmread('train_X.mtx').tocsr(), mmread('test_X.mtx').tocsr(), mmread('val_X.mtx').tocsr()
y_train, y_test, y_val = mmread('train_y.mtx').tocsr(), mmread('test_y.mtx').tocsr(), mmread('val_y.mtx').tocsr()

y = vstack((y_train, y_test, y_val))
X = vstack((X_train, X_test, X_val))

# perform a label powerset transformation on the label matrix y

transformer = LabelPowerset()
y_trans = transformer.transform(y)

new_y_train = y_trans[0:y_train.shape[0],]
new_y_test  = y_trans[y_train.shape[0]:(y_train.shape[0] + y_test.shape[0]),]
new_y_val   = y_trans[(y_train.shape[0] + y_test.shape[0]):y_trans.shape[0], ]



def RF_predict(n_estimators, max_depth, min_samples_split, X_train, y_train, X_test, X_val, n_jobs):

        # function which trains a random forest model and outputs predictions
        # of the test and validation sets

        # inputs: n_estimators      - number of learners in the random forest
        #         max_depth         - maximum tree depth of the random forest
        #         min_samples_split - the minimum number of samples to split on in the random forest
        #         X_train           - training data
        #         y_train           - train label data after having been transformed to the label powerset
        #         X_test            - test data
        #         X_val             - validation data
        #         n_jobs            - number of cores to use

        # outputs: test_preds - the predictions for the test set
        #          val_preds  - the predictions for the validation set

        classifier = RandomForestClassifier(n_estimators = n_estimators, max_depth = max_depth,
                                            min_samples_split = min_samples_split, verbose = 2, n_jobs = n_jobs)

        classifier.fit(X_train, y_train)
        test_pred = classifier.predict(X_test)
        val_pred  = classifier.predict(X_val)

        return(test_pred, val_pred)

def Extra_predict(n_estimators, max_depth, min_samples_split, X_train, y_train, X_test, X_val, n_jobs):

        # function which trains an extremely randomised trees classifier and outputs predicitons on
        # the test and validation sets.

        # inputs: n_estimators      - number of learners in the ensemble
        #         max_depth         - maximum tree depth of each tree
        #         min_samples_split - the minimum number of samples to split on
        #         X_train           - training data
        #         y_train           - train label data after having been transformed to the label powerset
        #         X_test            - test data
        #         X_val             - validation data
        #         n_jobs            - number of cores to use

        # outputs: test_preds - the predictions for the test set
        #          val_preds  - the predictions for the validation set

        classifier = ExtraTreesClassifier(n_estimators = n_estimators, max_depth = max_depth,
                                          min_samples_split = min_samples_split, verbose = 2, n_jobs = n_jobs)

        classifier.fit(X_train, y_train)
        test_pred = classifier.predict(X_test)
        val_pred  = classifier.predict(X_val)

        return(test_pred, val_pred)


def save_RF(n_estimators, max_depth, min_samples_split, X_train, y_train, X_test, X_val, n_jobs):

    # this function saves the predictions of a random forest classifier

    # inputs: n_estimators      - number of learners in the ensemble
    #         max_depth         - maximum tree depth of each tree
    #         min_samples_split - the minimum number of samples to split on
    #         X_train           - training data
    #         y_train           - train label data after having been transformed to the
    #                             label powerset
    #         X_test            - test data
    #         X_val             - validation data
    #         n_jobs              - number of cores to use

    test_preds, val_preds = RF_predict(n_estimators = n_estimators, max_depth = max_depth,
                                       min_samples_split = min_samples_split,
                                       X_train = X_train, y_train = y_train, X_test = X_test,
                                       X_val = X_val, n_jobs = n_jobs)

    print(accuracy_score(test_preds, new_y_test))
    test_filename = "RF_test_{}_{}_{}_{}".format(n_estimators, max_depth, min_samples_split)
    val_filename = "RF_val_{}_{}_{}_{}".format(n_estimators, max_depth, min_samples_split)
    np.save(test_filename, test_preds)
    np.save(val_filename, val_preds)

def save_Extra(n_estimators, max_depth, min_samples_split, X_train, y_train, X_test, X_val,
     n_jobs):

    # this function saves the predictions of a random forest classifier

    # inputs: n_estimators       - number of learners in the ensemble
    #         max_depth          - maximum tree depth of each tree
    #         min_samples_split  - the minimum number of samples to split on
    #         X_train            - training data
    #         y_train            - train label data after having been transformed to the
    #                               label powerset
    #         X_test             - test data
    #         X_val              - validation data
    #         n_jobs             - number of cores to use

    test_preds, val_preds = Extra_predict(n_estimators = n_estimators, max_depth = max_depth,
                                       min_samples_split = min_samples_split,
                                       X_train = X_train, y_train = y_train, X_test = X_test,
                                       X_val = X_val, n_jobs = n_jobs)

    print(accuracy_score(test_preds, new_y_test))
    test_filename = "ET_test_{}_{}_{}_{}".format(n_estimators, max_depth, min_samples_split)
    val_filename = "ET_val_{}_{}_{}_{}".format(n_estimators, max_depth, min_samples_split)
    np.save(test_filename, test_preds)
    np.save(val_filename, val_preds)



# Now we train and save the predictions our base learners which will be combined in our model
     stacker


# loop for the shallow random forest

for i in range(60, 90, 10):
    for j in range(4000, 7000, 1000):
        for k in range(2, 18, 2):
            save_RF(n_estimators = i, max_depth = j, min_samples_split = k,
                    X_train = X_train, y_train = new_y_train, X_test = X_test,
                    X_val = X_val, n_jobs = 90)

# loop for the medium depth random forest

for i in range(18, 24, 3):
    for j in range(10000, 25000, 5000):
        for k in range(2, 17, 2):
            save_RF(n_estimators = i, max_depth = j, min_samples_split = k,
                    X_train = X_train, y_train = new_y_train,
                    X_test = X_test, X_val = X_val, n_jobs = 90)


# loop for the deep random forests

for i in range(4, 8):
    for j in range(40000, 70000, 10000):
        for k in range(2, 18, 2):
            save_RF(n_estimators = i, max_depth = j, min_samples_split = k,
                    X_train = X_train, y_train = new_y_train, X_test = X_test,
                    X_val = X_val, n_jobs = 90)

# loop for the deep extra trees

for i in range(7, 10):
    for j in range(20000, 50000, 10000):
        for k in range(2, 18, 2):
            save_Extra(n_estimators = i, max_depth = j, min_samples_split = k,
                       X_train = X_train, y_train = new_y_train, X_test = X_test,
                       X_val = X_val, n_jobs = 90)

# loop for the shallow extra trees

for i in range(15, 30, 5):
    for j in range(5000, 15000, 2500):
        for k in range(2, 18, 2):
            save_Extra(n_estimators = i, max_depth = j, min_samples_split = k,
                      X_train = X_train, y_train = new_y_train,
                      X_test = X_test, X_holdout = X_val, n_jobs = 90)
```

Listing B.6: Train Base Models

```

import pickle
import pandas as pd
import numpy as np
from numpy.random import seed
#from functools import partial

from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from skmultilearn.problem_transform import LabelPowerset

import GPyOpt
from GPyOpt.experiment_design import initial_design
#from xgboost import XGBClassifier

from scipy.io import mmread
from scipy.sparse import csr_matrix, vstack, hstack

import numpy as np
import matplotlib.pyplot as plt

import time
from scipy import stats

import matplotlib.mlab as mlab

import numpy as np
import pickle
from functools import partial
from itertools import chain

##### in this section we tune the hyperparameters of our base model using Bayesian optimisation

# load data

X_train, X_test, X_val = mmread('train_X.mtx').tocsr(), mmread('test_X.mtx').tocsr(), mmread('
    val_X.mtx').tocsr()
y_train, y_test, y_val = mmread('train_y.mtx').tocsr(), mmread('test_y.mtx').tocsr(), mmread('
    val_y.mtx').tocsr()

y = vstack((y_train, y_test, y_val))
X = vstack((X_train, X_test, X_val))

# label powerset transformation

transformer = LabelPowerset()
y_trans = transformer.transform(y)

new_y_train = y_trans[0:y_train.shape[0],]
new_y_test  = y_trans[y_train.shape[0]:(y_train.shape[0] + y_test.shape[0]),]
new_y_val   = y_trans[(y_train.shape[0] + y_test.shape[0]):y_trans.shape[0], ]

## now we declare the domain over which to search for the Bayesian optimisation.

domain = [
          ####Three Random Forests
          ### shallow forest

          ## Number of estimators: optimizing the noise&variance
          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(60, 90, 10)},
          ##Maximum depth of the tree: how complex is the output
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(4000, 7000, 1000)},
          ##Minimum number of samples for a node to be split:
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)}

          ### medium depth forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(18, 24, 3)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(10000, 25000, 5000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 17, 2)},

          ## deep forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(4, 8)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(40000, 70000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          #### Extremely Randomised Trees
          # deep extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(6, 10)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(20000, 50000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          ####Three Random Forests
          # shallow forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(60, 90, 10)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(4000, 7000, 1000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          # medium depth forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(18, 24, 3)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(10000, 25000, 5000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 17, 2)},

          # deep forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(4, 8)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(40000, 70000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},


          #####Two Extremely Randomised Trees
          # deep extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(6, 10)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(20000, 50000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          # shallow extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(10, 30, 5)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(5000, 15000, 2500)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          ####################################### Stacker

          {'name': 'stacker', 'type': 'discrete', 'domain': range(2)},
          # 0 for logistic regression, 1 for majority voting

          ############################### Drop different base learners

          {'name': 'shallow_rf', 'type': 'discrete', 'domain': range(2)},
          {'name': 'med_rf', 'type': 'discrete', 'domain': range(2)},
          {'name': 'deep_rf', 'type': 'discrete', 'domain': range(2)},
          {'name': 'deep_et', 'type': 'discrete', 'domain': range(2)}
          {'name': 'shallow_rf_2', 'type': 'discrete', 'domain': range(2)},
          {'name': 'med_rf_2', 'type': 'discrete', 'domain': range(2)},
          {'name': 'deep_rf_2', 'type': 'discrete', 'domain': range(2)},
          {'name': 'deep_et_2', 'type': 'discrete', 'domain': range(2)}
          {'name': 'shallow_et_2', 'type': 'discrete', 'domain': range(2)}
          ]

#### Here we define the objective function


def objective_function(hyperparameters, y_test, y_val, transformer):

    # objective function to optimise using Bayesian optimisation

    # load the test predictions of the base learners

    hyperparameters = np.array(hyperparameters)

    n = 1
    test_models = []
    val_models  = []

    print('loading models')

    if hyperparameters[0][28] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_2.npy'.format(hyperparameters[0][0].astype(int)
        , hyperparameters[0][1].astype(int), hyperparameters[0][2].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_2.npy'.format(hyperparameters[0][0].astype(int)
        , hyperparameters[0][1].astype(int), hyperparameters[0][2].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][29] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][3].astype(int)
        , hyperparameters[0][4].astype(int), hyperparameters[0][5].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][3].astype(int)
        , hyperparameters[0][4].astype(int), hyperparameters[0][5].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][30] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][6].astype(int)
        , hyperparameters[0][7].astype(int), hyperparameters[0][8].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][6].astype(int)
        , hyperparameters[0][7].astype(int), hyperparameters[0][8].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][31] == 1:
        test_loaded = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][9].astype(int)
        , hyperparameters[0][10].astype(int), hyperparameters[0][11].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][9].astype(int)
        , hyperparameters[0][10].astype(int), hyperparameters[0][11].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][32] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_2.npy'.format(hyperparameters[0][12].astype(int
    )
        , hyperparameters[0][13].astype(int), hyperparameters[0][14].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_2.npy'.format(hyperparameters[0][12].astype(int)
        , hyperparameters[0][13].astype(int), hyperparameters[0][14].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][33] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][15].astype(int
    )
        , hyperparameters[0][16].astype(int), hyperparameters[0][17].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][15].astype(int)
        , hyperparameters[0][16].astype(int), hyperparameters[0][17].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][34] == 1:
        test_loaded = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][18].astype(int
    )
        , hyperparameters[0][19].astype(int), hyperparameters[0][20].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][18].astype(int)
        , hyperparameters[0][19].astype(int), hyperparameters[0][20].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    if hyperparameters[0][35] == 1:
        test_loaded = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][21].astype(int
    )
        , hyperparameters[0][22].astype(int), hyperparameters[0][23].astype(int)))
        test_models.append(test_loaded)

        val_loaded = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][21].astype(int)
        , hyperparameters[0][22].astype(int), hyperparameters[0][23].astype(int)))
        val_models.append(val_loaded)
        n = n + 1

    test_loaded = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][24].astype(int)
    , hyperparameters[0][25].astype(int), hyperparameters[0][26].astype(int)))
    val_loaded = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][24].astype(int)
    , hyperparameters[0][25].astype(int), hyperparameters[0][26].astype(int)))

    test_models.append(test_loaded)
    val_models.append(val_loaded)


    # now we choose a stacker

    print('models loaded')

    if hyperparameters[0][27] == 0:

        # logistic regression stacker

        test = []
        val  = []

        for i in range(n):
            test_trans = transformer.inverse_transform(test_models[i])
            val_trans  = transformer.inverse_transform(val_models[i])

            test.append(test_trans)
            val.append(val_trans)

        S_train = test[0]
        S_test  = val[0]

        if n > 1:
            for i in range(n - 1):
                S_train = hstack((S_train, test[i + 1]))
                S_test  = hstack((S_test, val[i + 1]))


        classifier = LogisticRegression()
        start_time = time.time()
        print('training model stacker with {} base models'.format(n))
        classifier.fit(S_train, y_test)

        training_time = time.time() - start_time
        print('model trained in {} seconds, starting prediction'.format(training_time))

        start_prediction = time.time()
        predictions = classifier.predict(S_test)
        print('prediction time: {}'.format(time.time() - start_prediction))

    else:

        # majority voting

        full_val = np.array(val_models)
        predictions = stats.mode(full_val)[0][0]

    accuracy = accuracy_score(predictions, y_val)
    print('accuracy: {}'.format(accuracy))
    print('hyperparameters: {}'.format(hyperparameters[0].astype(int)))

    return(accuracy)


## we tried various combinations and found that 7 base learners - three random forest
## and four extremely randomised trees, stacked using majority voting, performed best.
## We therefore run a smaller optimisation using just these learners in order to tune
## their hyperparameters more directly.

domain_7 = [
          ####Three Random Forests
          ### shallow forest

          ## Number of estimators: optimizing the noise&variance
          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(60, 90, 10)},
          ##Maximum depth of the tree: how complex is the output
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(4000, 7000, 1000)},
          ##Minimum number of samples for a node to be split:
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)}

          ### medium depth forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(18, 24, 3)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(10000, 25000, 5000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 17, 2)},

          ## deep forest

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(4, 8)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(40000, 70000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          #### Extremely Randomised Trees
          # deep extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(6, 10)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(20000, 50000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          # shallow extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(10, 30, 5)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(5000, 15000, 2500)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          # deep extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(6, 10)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(20000, 50000, 10000)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)},

          # shallow extra trees

          {'name': 'n_estimators', 'type': 'discrete', 'domain': range(10, 30, 5)},
          {'name': 'max_depth', 'type': 'discrete', 'domain': range(5000, 15000, 2500)},
          {'name': 'min_samples_split', 'type': 'discrete', 'domain': range(2, 18, 2)}

]

def objective_function_7(hyperparameters, y_test, y_val, transformer):

    # This is the objective function which the Bayesian optimisation will optimise.

    # inputs: hyperparameters - the hyperparameters of the base models.
    #                           will determine the values of these.
    #         y_test          - the transformed test set labels
    #         y_val           - the transformed validation set labels
    #         transformer     - the label powerset transformer


    hyperparameters = np.array(hyperparameters)
    test_models = []
    val_models  = []
    print('loading models')

    # load the predictions

    shallow_rf_test = np.load('RF_test_{}_{}_{}_2.npy'.format(hyperparameters[0][0].astype(int)
    , hyperparameters[0][1].astype(int), hyperparameters[0][2].astype(int)))
    med_rf_test = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][3].astype(int)
    , hyperparameters[0][4].astype(int), hyperparameters[0][5].astype(int)))
    deep_rf_test = np.load('RF_test_{}_{}_{}_1.npy'.format(hyperparameters[0][6].astype(int)
    , hyperparameters[0][7].astype(int), hyperparameters[0][8].astype(int)))

    deep_et_test = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][9].astype(int)
    , hyperparameters[0][10].astype(int), hyperparameters[0][11].astype(int)))
    shallow_et_test = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][12].astype(int
     )
    , hyperparameters[0][13].astype(int), hyperparameters[0][14].astype(int)))

    shallow_rf_val = np.load('RF_val_{}_{}_{}_2.npy'.format(hyperparameters[0][0].astype(int)
    , hyperparameters[0][1].astype(int), hyperparameters[0][2].astype(int)))
    med_rf_val = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][3].astype(int)
    , hyperparameters[0][4].astype(int), hyperparameters[0][5].astype(int)))
    deep_rf_val = np.load('RF_val_{}_{}_{}_1.npy'.format(hyperparameters[0][6].astype(int)
    , hyperparameters[0][7].astype(int), hyperparameters[0][8].astype(int)))

    deep_et_val = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][9].astype(int)
    , hyperparameters[0][10].astype(int), hyperparameters[0][11].astype(int)))
    shallow_et_val = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][12].astype(int)
    , hyperparameters[0][13].astype(int), hyperparameters[0][14].astype(int)))

    deep_et_test_2 = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][15].astype(int)
    , hyperparameters[0][16].astype(int), hyperparameters[0][17].astype(int)))
    shallow_et_test_2 = np.load('ET_test_{}_{}_{}_1.npy'.format(hyperparameters[0][18].astype(
     int)
    , hyperparameters[0][19].astype(int), hyperparameters[0][20].astype(int)))

    deep_et_val_2 = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][15].astype(int)
    , hyperparameters[0][16].astype(int), hyperparameters[0][17].astype(int)))
    shallow_et_val_2 = np.load('ET_val_{}_{}_{}_1.npy'.format(hyperparameters[0][18].astype(int
     )
    , hyperparameters[0][19].astype(int), hyperparameters[0][20].astype(int)))

    print('models loaded')


     # majority voting

    full_val = np.array([shallow_rf_val, med_rf_val, deep_rf_val, deep_et_val,
    shallow_et_val, deep_et_val_2, shallow_et_val_2])
    predictions = stats.mode(full_val)[0][0]

    val_accuracy = accuracy_score(predictions, y_val)
    print('val_accuracy: {}'.format(val_accuracy))
    print('hyperparameters: {}'.format(hyperparameters[0].astype(int)))

    return(accuracy)


# let the objective function only be a function of the hyperparameters
f_7 = partial(objective_function_7, y_test = new_y_test,
                y_val = new_y_val, transformer = transformer)

batch_size = 16
num_cores  = 16

# set some initial points based around some of the best base learners

initial_points = np.array([[80, 5000, 16, 21, 20000, 6, 7, 60000, 2, 9, 20000, 4, 25, 12500,
                             2, 9, 20000, 8, 20, 12500, 4],
                            [80, 6000, 2, 18, 20000, 4, 4, 60000, 10, 7, 40000, 6, 20, 10000,
                            4, 7, 20000, 6, 25, 10000, 6],
                            [80, 6000, 10, 21, 20000, 2, 7, 50000, 6, 9, 20000, 2, 20, 12500,
                            4, 6, 20000, 4, 25, 12500, 4],
                            [60,  6000, 6, 21, 10000, 4, 7, 40000, 4, 9, 40000, 6, 25, 10000,
                            4, 9, 20000, 4, 20, 10000, 2],
                            [60, 6000, 8, 21, 20000, 6, 7, 50000, 10, 7, 20000, 4, 20, 12500,
                            2, 9, 20000, 4, 25, 12500, 4],
                            [80, 6000, 10, 18, 20000, 4, 4, 60000, 10, 9, 20000, 6, 25, 10000,
                            2, 8, 20000, 2, 20, 12500, 2],
                            [60, 6000, 12, 18, 20000, 8, 4, 50000, 10, 7, 40000, 6, 20, 10000,
                            6, 9, 20000, 6, 25, 10000, 4],
                            [70, 6000, 14, 18, 20000, 12, 4, 50000, 10, 7, 40000, 2, 20, 10000,
                            2, 7, 20000, 8, 25, 10000, 8],])

# here we initialise the optimisation

BO_7 =  methods.BayesianOptimization(f = f_7,                       # Objective function
                                     domain = domain_7,             # domain
                                     initial_design_numdata = 8,    # number of initial points
                                     acquisition_type='EI',         # Expected Improvement
                                     evaluator_type = 'local_penalization', # parallel
                                     batch_size = batch_size,
                                     maximize = True,
                                     X = initial_points,
                                            num_cores = num_cores,
                                            exact_feval = False)            # noisy evaluations

# now we run the optimisation and save the reports and models every 10 iterations

for i in range(100):

    max_iter = 10
    BO_7.run_optimization(max_iter)

    BO_7.save_models('reports/7_saved_model_step_%d'%i)
    BO_7.save_report('reports/7_saved_report_step_%d'%i)
    BO_7.save_evaluations('reports/7_saved_evaluations_step_%d'%i)
```

Listing B.7: Bayesian Optimisation

References

[1] Primo Levi and Raymond Rosenthal. *The periodic table*. Penguin, 2012.

[2] Isaac Asimov. *I, robot*. Harper Voyager, 1950.

[3] Carl Benedikt Frey and Michael A. Osborne. "The Future of Employment: How Susceptible Are Jobs to Computerisation?" In: *Oxford Martin* 114 (Jan. 2013).

[4] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. URL: `http://www.sciencedirect.com/science/article/pii/S0004370201001291`.

[5] Garry Kasparov. *The Chess Master and the Computer*. URL: `https://www.nybooks.com/articles/2010/02/11/the-chess-master-and-the-computer/?pagination=false`.

[6] Neil Rubens, Dain Kaplan, and Masashi Sugiyama. "Active Learning in Recommender Systems". In: *Recommender Systems Handbook* (2010), pp. 735–767.

[7] Willem van Hoorn. *Searching chemical structures in patents using Bayesian statistics*. Sept. 2017. URL: `https://www.slideshare.net/vanhoorn/searching-chemical-structures-in-patents-using-bayesian-statistics-biovia-european-user-forum-2017`.

[8] Benjamin E. Blass. *Basic Principles of Drug Discovery and Development*. eng. Amsterdam, 2015.

[9] Andrew R. Leach and Valerie J. Gillet. *An Introduction To Chemoinformatics*. Springer, 2007.

[10] David Weininger. "SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules". In: *Journal of Chemical Information and Modeling* 28.1 (1988), pp. 31–36.

[11] H. L. Morgan. "The Generation of a Unique Machine Description for Chemical Structures-A Technique Developed at Chemical Abstracts Service." In: *Journal of Chemical Documentation* 5.2 (1965), pp. 107–113.

[12] David Rogers and Mathew Hahn. "Extended-Connectivity Fingerprints". In: *Journal of Chemical Information and Modeling* 50.5 (2010), pp. 742–754.

[13] George Papadatos et al. "SureChEMBL: a large-scale, chemically annotated patent document database". In: *Nucleic Acids Research* 44.D1 (2015).

[14] Daniel. *Unleashing over a million reactions into the wild*. URL: `https://nextmovesoftware.com/blog/2014/02/27/unleashing-over-a-million-reactions-into-the-wild/`.

[15] D. M. Lowe. "Extraction of chemical structures and reactions from the literature". PhD thesis. University of Cambridge, 2012.

[16] *Reaction SMILES and SMIRKS*. URL: `https://www.daylight.com/meetings/summerschool01/course/basics/smirks.html`.

[17] Nic Fleming. "How artificial intelligence is changing drug discovery". In: *Nature* 557.7707 (2018).

[18] Yu-Chen Lo et al. “Machine learning in chemoinformatics and drug discovery”. In: *Drug Discovery Today* 23.8 (2018), pp. 1538–1546.

[19] George Papadatos et al. “Activity, assay and target data curation and quality in the ChEMBL database”. In: *Journal of Computer-Aided Molecular Design* 29.9 (Sept. 2015), pp. 885–896.

[20] Hongming Chen et al. “The rise of deep learning in drug discovery”. In: *Drug Discovery Today* 23.6 (2018), pp. 1241–1250. URL: http://www.sciencedirect.com/science/article/pii/S1359644617303598.

[21] David K Duvenaud et al. “Convolutional Networks on Graphs for Learning Molecular Fingerprints”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2224–2232. URL: http://papers.nips.cc/paper/5954-convolutional-networks-on-graphs-for-learning-molecular-fingerprints.pdf.

[22] Esben Jannik Bjerrum. “SMILES Enumeration as Data Augmentation for Neural Network Modeling of Molecules”. In: *CoRR* abs/1703.07076 (2017). arXiv: 1703.07076.

[23] Rafael Gómez-Bombarelli et al. “Automatic chemical design using a data-driven continuous representation of molecules”. In: *CoRR* abs/1610.02415 (2016).

[24] Matt J. Kusner, Brooks Paige, and José Miguel Hernández-Lobato. “Grammar Variational Autoencoder”. In: *ICML*. 2017.

[25] Tao Huang et al. “MOST: most-similar ligand based approach to target prediction”. In: *BMC Bioinformatics* 18.1 (2017).

[26] Roberta G. Susnow and Steven L. Dixon. “Use of Robust Classification Techniques for the Prediction of Human Cytochrome P450 2D6 Inhibition.” In: *ChemInform* 34.42 (2003).

[27] Vladimir Svetnik et al. “Random Forest: A Classification and Regression Tool for Compound Classification and QSAR Modeling”. In: *Journal of Chemical Information and Computer Sciences* 43.6 (2003), pp. 1947–1958.

[28] Min Shen et al. “Development and Validation ofk-Nearest-Neighbor QSPR Models of Metabolic Stability of Drug Candidates”. In: *Journal of Medicinal Chemistry* 46.14 (2003), pp. 3013–3020.

[29] Mohammad A. Khanfar and Mutasem O. Taha. “Elaborate Ligand-Based Modeling Coupled with Multiple Linear Regression and k Nearest Neighbor QSAR Analyses Unveiled New Nanomolar mTOR Inhibitors”. In: *Journal of Chemical Information and Modeling* 53.10 (2013), pp. 2587–2612.

[30] Xiaoyang Xia et al. “Classification of Kinase Inhibitors Using a Bayesian Model”. In: *Journal of Medicinal Chemistry* 47.18 (2004), pp. 4463–4470.

[31] Lucian Chan, Geoffrey Hutchison, and Garrett Morris. “Bayesian Optimization for Conformer Generation”. In: *Journal of Cheminformatics* 11 (May 2019).

[32] Leonidas Aristodemou and Frank Tietze. “The state-of-the-art on Intellectual Property Analytics (IPA): A literature review on artificial intelligence, machine learning and deep learning methods for analysing intellectual property (IP) data”. In: *World Patent Information* 55 (2018), pp. 37–51.

[33] Fujin Zhu et al. “A Supervised Requirement-oriented Patent Classification Scheme Based on the Combination of Metadata and Citation Information”. In: *International Journal of Computational Intelligence Systems* 8.3 (2015), pp. 502–516.

[34] Sunghae Jun, Sang-Sung Park, and Dong-Sik Jang. “Document clustering method using dimension reduction and support vector clustering to overcome sparseness”. In: *Expert Systems with Applications* 41.7 (2014), pp. 3204–3212.

[35] Karol Molga, Piotr Dittwald, and Bartosz A. Grzybowski. “Navigating around Patented Routes by Preserving Specific Motifs along Computer-Planned Retrosynthetic Pathways”. In: *Chem* 5.2 (2019), pp. 460–473.

[36] Grigorios Tsoumakas and Ioannis Katakis. “Multi-Label Classification”. In: *International Journal of Data Warehousing and Mining* 3.3 (2007), pp. 1–13.

[37] Kush Bhatia et al. “Sparse Local Embeddings for Extreme Multi-label Classification”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 730–738.

[38] Wei Bi and James T. Kwok. “Efficient Multi-label Classification with Many Labels”. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML’13. Atlanta, GA, USA: JMLR.org, 2013, pp. III-405–III-413.

[39] Yao-nan Chen and Hsuan-tien Lin. “Feature-aware Label Space Dimension Reduction for Multi-label Classification”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1529–1537.

[40] Rahul Agrawal et al. “Multi-label learning with millions of labels”. In: *Proceedings of the 22nd international conference on World Wide Web - WWW 13* (2013).

[41] Yashoteja Prabhu and Manik Varma. “FastXML”. In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 14* (2014).

[42] Daniel J Hsu et al. “Multi-Label Prediction via Compressed Sensing”. In: *Advances in Neural Information Processing Systems 22*. Ed. by Y. Bengio et al. Curran Associates, Inc., 2009, pp. 772–780.

[43] Moustapha M Cisse et al. “Robust Bloom Filters for Large MultiLabel Classification Tasks”. In: *Advances in Neural Information Processing Systems 26*. Ed. by C. J. C. Burges et al. Curran Associates, Inc., 2013, pp. 1851–1859.

[44] F. Tai and H. Lin. “Multilabel Classification with Principal Label Space Transformation”. In: *Neural Computation* 24.9 (Sept. 2012), pp. 2508–2542.

[45] Prateek Jain, Raghu Meka, and Inderjit S. Dhillon. “Guaranteed Rank Minimization via Singular Value Projection”. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 937–945.

[46] Jason Weston, Ameesh Makadia, and Hector Yee. “Label Partitioning For Sublinear Ranking”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 2. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 181–189. URL: `http://proceedings.mlr.press/v28/weston13.html`.

[47] Yining Wang et al. *A Theoretical Analysis of NDCG Type Ranking Measures*. 2013. arXiv: `1304.6480 [cs.LG]`.

[48] Jesse Read et al. “Classifier chains for multi-label classification”. In: *Machine Learning* 85.3 (June 2011), p. 333.

[49] Eneldo Loza Mencía, Sang-Hyeun Park, and Johannes Fürnkranz. “Efficient voting prediction for pairwise multilabel classification”. In: *Neurocomputing* 73.7 (2010). Advances in Computational Intelligence and Learning, pp. 1164–1176.

[50] Matthew R. Boutell et al. "Learning multi-label scene classification". In: *Pattern Recognition* 37.9 (2004), pp. 1757–1771.

[51] P. Szymański and T. Kajdanowicz. "A scikit-based Python environment for performing multi-label classification". In: *ArXiv e-prints* (Feb. 2017). arXiv: `1702.01460 [cs.LG]`.

[52] Yamin Sun, Andrew K. C. Wong, and Mohammed S. Kamel. "Classification of Imbalanced Data: A Review". In: *International Journal of Pattern Recognition and Artificial Intelligence* 23.04 (2009), pp. 687–719.

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

[54] *RDKit*. `https://www.rdkit.org/`. (Accessed on 07/27/2019).

[55] The GPyOpt authors. *GPyOpt: A Bayesian Optimization framework in Python*. `http://github.com/SheffieldML/GPyOpt`. 2016.

[56] Robert B. Heywood and Mortimer J. Adler. *The Works of the Mind*. 1966.

[57] Raphael Yuster and Uri Zwick. "Fast Sparse Matrix Multiplication". In: *Algorithms – ESA 2004*. Ed. by Susanne Albers and Tomasz Radzik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 604–615.

[58] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288.

[59] G.h. Golub, Alan Hoffman, and G.w. Stewart. "A generalization of the Eckart-Young-Mirsky matrix approximation theorem". In: *Linear Algebra and its Applications* 88-89 (1987), pp. 317–327.

[60] Andrew Y. Ng and Michael I. Jordan. "On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes". In: *Advances in Neural Information Processing Systems 14*. Ed. by T. G. Dietterich, S. Becker, and Z. Ghahramani. MIT Press, 2002, pp. 841–848.

[61] V. N. Vapnik. *Statistical learning theory*. J. Wiley, 1998.

[62] Ahmad Ashari, Iman Paryudi, and A Min. "Performance Comparison between Naïve Bayes, Decision Tree and k-Nearest Neighbor in Searching Alternative Design in an Energy Simulation Tool". In: *International Journal of Advanced Computer Science and Applications* 4.11 (2013).

[63] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. "Spam Filtering with Naive Bayes - Which Naive Bayes?" In: Jan. 2006.

[64] Pier Francesco Palamara. *Statistical Machine Learning*. Feb. 2019. URL: `https://www.stats.ox.ac.uk/~palamara/teaching/SML19/HT19_lecture11.pdf`.

[65] Andrew McCallum and Kamal Nigam. "A Comparison of Event Models for Naive Bayes Text Classification". In: 1998.

[66] X Han. *Python implementation of "Sparse Local Embeddings for Extreme Multi-label Classification, NIPS, 2015"*. `https://github.com/xiaohan2012/sleec_python`. 2017.

[67] Y. Hu, Y. Koren, and C. Volinsky. "Collaborative Filtering for Implicit Feedback Datasets". In: *2008 Eighth IEEE International Conference on Data Mining*. Dec. 2008, pp. 263–272.

[68] S Boyd. *Alternating Direction Method of Multipliers*. URL: `https://web.stanford.edu/~boyd/papers/pdf/admm_slides.pdf`.

[69] *Multi-label Datasets Code*. URL: `http://manikvarma.org/downloads/XC/XMLRepository.html#Prabhu14`.

[70] Joseph Sill et al. "Feature-Weighted Linear Stacking". In: *CoRR* abs/0911.0460 (2009). arXiv: `0911.0460`.

[71] Thomas G. Dietterich. "Ensemble Methods in Machine Learning". In: *Multiple Classifier Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–15.

[72] Linhua Wang et al. "Large-scale protein function prediction using heterogeneous ensembles". In: *F1000Research* 7 (2018), p. 1577.

[73] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32.

[74] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine Learning* 63.1 (Apr. 2006), pp. 3–42.

[75] Lior Rokach and Oded Maimon. *Data Mining With Decision Trees: Theory and Applications*. 2nd. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2014.

[76] Gilles Louppe. "Understanding Random Forests: From Theory to Practice". arXiv:1407.7502. PhD thesis. University of Liege, Belgium, Oct. 2014.

[77] Leo Breiman. "Bagging predictors". In: *Machine Learning* 24.2 (Aug. 1996), pp. 123–140.

[78] B. Efron. "Bootstrap Methods: Another Look at the Jackknife". In: *Ann. Statist.* 7.1 (Jan. 1979), pp. 1–26.

[79] Manuel Fernandez-Delgado et al. "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?" In: *Journal of Machine Learning Research* 15 (Oct. 2014), pp. 3133–3181.

[80] Thanh-Nghi Do et al. "Classifying Very-High-Dimensional Data with Random Forests of Oblique Decision Trees". In: *Advances in knowledge discovery and management* (Jan. 2009).

[81] Robert E. Schapire. "The strength of weak learnability". In: *Machine Learning* 5.2 (June 1990), pp. 197–227.

[82] Yoav Freund and Robert E. Schapire. "A Short Introduction to Boosting". In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1999, pp. 1401–1406.

[83] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *CoRR* abs/1603.02754 (2016). arXiv: `1603.02754`.

[84] J Mockus, Vytautas Tiesis, and Antanas Zilinskas. "The application of Bayesian methods for seeking the extremum". In: vol. 2. Sept. 2014, pp. 117–129.

[85] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: *CoRR* abs/1012.2599 (2010). arXiv: `1012.2599`.

[86] Olivier Bousquet, Ulrike von. Luxburg, and Rätsch Gunnar. *Advanced lectures on machine learning: ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003: revised lectures*. Springer, 2004.

[87] Dino Sejdinovic. *Advanced Topics in Statistical Machine Learning*. Feb. 2019. URL: `https://www.stats.ox.ac.uk/~sejdinov/atsml19/`.

[88] Morris L. Eaton. *Multivariate statistics: a vector space approach*. John Wiley and Sons, 1983.

[89] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *arXiv e-prints*, arXiv:1206.2944 (June 2012), arXiv:1206.2944.

[90] Paul Valéry. *Collected Works of Paul Valéry*. 1970.

[91] *BIOVIA Pipeline Pilot.* URL: `https://www.3dsbiovia.com/products/collaborative-science/biovia-pipeline-pilot/`.

[92] Xiaoyang Xia et al. “Classification of Kinase Inhibitors Using a Bayesian Model”. In: *Journal of Medicinal Chemistry* 47.18 (2004). PMID: 15317458, pp. 4463–4470.

[93] Paul Bennett. “Assessing the Calibration of Naive Bayes’ Posterior Estimates”. In: (Oct. 2000).

[94] Thomas Carlyle. *Oliver Cromwells Letters and speeches.* Harper, 1855.

[95] *AI Platform Notebooks | Google Cloud.* URL: `https://cloud.google.com/ai-platform-notebooks/`.

[96] Habshah Midi, S.K. Sarkar, and Sohel Rana. “Collinearity diagnostics of binary logistic regression model”. In: *Journal of Interdisciplinary Mathematics* 13.3 (2010), pp. 253–267.

[97] Marcus Tullius Cicero. *De finibus bonorum et malorum.* 45BCE.