

GENerateZ: Automatic *De Novo* Design of Anticancer Drugs using Transcriptomic Data, Genetic Algorithms, and Variational Autoencoders



Hans William Alexander Hanley

Word count: 11, 972

Submitted in partial completion of the
Master of Science in Statistical Science

I hereby certify that this is entirely
my own work unless otherwise stated.

Michaelmas 2020

Acknowledgements

I would like to thank Professor Garrett M. Morris for guiding me throughout the course of this work. Without his feedback, suggestions, and help, this work would have not been possible.

I would also like to thank my parents and siblings: Patricia, Allison, Samantha, Jonathan, Ashley, and Arielle Hanley whose support throughout the pandemic enabled me to stay sane and focus on this project.

Lastly, I would like to thank the Daniel M. Sachs scholarship for funding me during my time at Oxford. Its support was invaluable to allowing me to complete this work.

Abstract

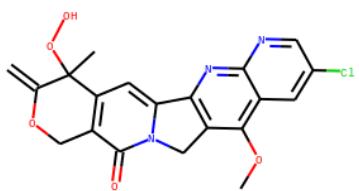
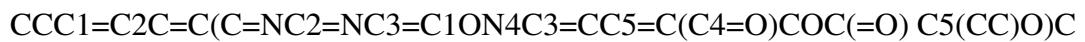
We propose a novel machine learning architecture and technique for *de novo* drug discovery of anti-cancer drugs by using discrete representations of drugs' chemical compositions and the transcriptomics of targets. In particular, we generate novel compounds optimized for high efficacy against specific types of cancerous cells.

First, we train a variational autoencoder (VAE) to convert discrete representations (SELFIES) of drug-like molecules to and from a multidimensional latent space. By utilizing the advanced scheduling technique cyclical annealing proposed by Fu et al., we avoid VAE posterior collapse. This advancement allows the extraction of highly meaningful latent features from discrete representations. As proof of the latent space's meaningfulness, we discover the distribution of molecules in the latent space follows a gradient of desirable chemical properties (Quantitative Estimate of Drug-Likeness and Synthetic Accessibility); *i.e.* molecules with high values cluster in one region of the latent space, and molecules with low values cluster in another. These properties are important criterion for pre-clinical drug discovery and discovering this gradient allows for efficient optimization.

Second, we train a separate Transformer decoder to map pretrained latent embeddings of molecules back to their chemical representations. We show that this separate and more powerful decoder can more effectively and easily map diverse sets of latent representations back into compounds than the original decoder in the VAE. This essentially turns our VAE into a highly efficient feature extractor and allows for more exploration of the chemical space created by the VAE.

Last, having created a highly information-dense latent space model and an efficient decoder, we jointly train the same VAE model with a separate auxiliary network designed to predict drug-like molecules' efficacy IC_{50} against a particular subset of cancer cell targets from their latent representations and the transcriptomic profiles of cancer cells. This joint training causes the latent space to develop gradients for a particular drug-like molecules' efficacy (IC_{50}) against these particular targets. We discover that by utilizing this approach, we can accurately predict IC_{50} efficacy from VAE latent representations and transcriptomic data with an overall Pearson correlation of 86.78 on a test set. Using this approach, we show that by using Bayesian optimization and genetic algorithms, we can optimize drug-like molecules' efficacy against a particular cancer target. We illustrate the usefulness of this approach by generating hundreds of novel potent drug

candidates, optimized for efficacy against a group of sarcoma cells. We verify these novel candidate drugs by comparing them to existing compounds with known efficacy against corresponding cancer type. We give an example an optimized drug:



We wish for our approach, which leverages the latest improvements in text generation from natural language processing, to be a step toward improving success rates of targeted and personalized drug discovery against cancers.

Contents

List of Figures	vii
1 Introduction	1
1.1 Motivation	1
1.2 Related Work	2
1.3 Contribution	3
2 Background	6
2.1 Introduction	6
2.2 Drug Discovery	6
2.2.1 Desirable Drug Properties	7
2.2.2 Transcriptomics	10
2.2.3 Simplified Molecular Input Entry System (SMILES)	11
2.2.4 DeepSMILES	12
2.2.5 Self-Referencing Embedded Strings (SELFIES)	12
2.2.6 SPVec	12
2.3 Variational Autoencoders	13
2.3.1 Posterior Collapse	15
2.3.2 Cyclical KL Annealing	15
2.3.3 Implicit Latent Codes	16
2.4 Additional Machine Learning Algorithms	18
2.4.1 Machine Learning for Manipulating the VAE Latent Space	18
2.4.2 Machine Learning Algorithms for Predicting Chemical Toxicity	22
2.4.3 Optimization Algorithms	24
2.4.4 Bayesian Optimization	25
3 Experimental Setup	26
3.1 Introduction	26
3.2 Datasets	26
3.2.1 ChEMBL	26
3.2.2 ZINC	27
3.2.3 IC ₅₀ Sensitivity: GDSC	27
3.2.4 Toxicity: DrugBank, KEGG, TOXNET, T3DB	28
3.3 Machine Learning Setup	28

4 Experiments	29
4.1 Introduction	29
4.2 Modelling Compounds: Variational Autoencoders and Transformers	30
4.2.1 Decoding Rates	32
4.2.2 Tanimoto Similarities	34
4.2.3 Compounds Near Ibuprofen in Latent Space	35
4.2.4 QED, logP, and SA Scores in the Latent Space	36
4.3 Predicting $\log_{10}IC_{50}$: Attention Based Neural Networks	39
4.4 Optimizing the Latent Space for Predicting $\log_{10}IC_{50}$	44
4.5 Predicting Toxicity with Ensembles: Random Forests, Extra Trees, DARTs, Neural Nets, and SVMs	47
4.6 Optimizing IC_{50} and Creating Realistic Drugs	49
4.6.1 Bayesian Optimization (BO)	51
4.6.2 Genetic Algorithm (GA)	51
4.6.3 Drug-likeness and Synthesizability Optimization	52
4.6.4 IC_{50} Optimization Against the TE-12 Cell Line	55
5 Discussion	62
5.1 Modelling Compounds	62
5.2 Predicting Toxicity	62
5.3 Predicting $\log_{10}IC_{50}$	63
5.4 Optimizing Compounds	64
5.5 Google Colab Issues	65
Appendices	
A Additional Graphs	67
A.1 Tanimoto Similarities	67
B Selected Code	72
References	149

List of Figures

2.1	KL Regularization term in Cyclical Annealing. Figure from [40].	16
2.2	Diagram illustrating the network structure of the implicit latent model VAE. Figure from [17]. X_i are time-series inputs at time i , h_i are hidden states computed by the model for time step i . For details on multi-layer perceptrons (MLP), Long short-term memory (LSTM) see Section 2.4.	17
2.3	Sigmoid, Tanh, and ReLU. Figure from [59]	19
2.4	LSTM Cell. Figure from [42].	20
2.5	Multi-headed Attention. Figure from [64].	21
2.6	Decoder-Layer and resulting full Decoder. Figure from [1].	22
2.7	SVM with dividing hyperplane H and margin M for a two-dimensional dataset (x_1, x_2) . Figure from [33]. The left shows a completely separable dataset while the right shows an inseparable set.	23
4.1	Variational autoencoder and training process. x is the discrete input vector, μ are the mean values calculated for the latent dimensions, σ are the standard deviations for the latent dimensions and z is the sampled latent representation. Encoder and Decoder are neural networks. Shows an example for the cancer drug Doxorubicin.	30
4.2	Hexbin plots and distribution histograms of the Tanimoto similarities of 4000 random chemical compounds from the ChEMBL SMILES test set against a single compound after projecting their latent representations using linear PCA. Identical pairs of molecules have a Tanimoto similarity of 1 and are coloured yellow, while less similar molecules are green, and completely dissimilar molecules are dark blue.	35
4.3	Chemical compounds in the vicinity of the Ibuprofen with their Euclidean distance (ED) in the latent space and Tanimoto Similarity (TS).	35
4.4	Hexbin plots of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ChEMBL test set after projecting using linear PCA the latent representations of the cyclically annealed VAEs.	36
4.5	Hexbin plots of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ChEMBL test set after projecting using linear PCA the latent representations from implicit VAE models.	37

4.6	Hexpolot of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ZINC SMILES testing test after projecting latent representations using linear PCA.	38
4.7	Model workflow to predict IC ₅₀	39
4.8	Calculation of gene embeddings. Figure from [41]. Gene subset is concatenated with itself after one version is with put through linear layer and a Softmax layer to compute an attention distribution (α_i).	40
4.9	Calculation of contextual embeddings [41]. Gene attention output and encoded smiles are put through A contextual attention (CA) layer. The CA layer then outputs an attention distribution(α_i) over the SMILES encoding, in the context of transcriptomic profile.	41
4.10	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and SMILES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale.	41
4.11	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and DeepSMILES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale.	42
4.12	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and SELFIES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale.	42
4.13	Normalized [0,1] log ₁₀ IC ₅₀ values for chemical compounds after projecting using linear PCA against the UMC-11 cell line, a cell of a carcinoid-endocrine tumour affecting the lung.	44
4.14	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and SMILES latent embeddings with IC ₅₀ latent shaping. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale. . . .	45
4.15	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and DeepSMILES latent embeddings with IC ₅₀ latent shaping. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale.	46
4.16	Prediction of log ₁₀ IC ₅₀ using transcriptomic data and SELFIES latent embeddings with IC ₅₀ latent shaping. The model was fitted in log space. RMSE was calculated after normalizing log ₁₀ IC ₅₀ on a [0,1] scale.	46
4.17	Summary of the VAE-DeepSMILES embeddings for predicting toxicity.	48
4.18	Example of optimization of normalized [0,1] log ₁₀ IC ₅₀ in the latent space.	49
4.19	Distribution of QED and SA Scores of found cancer drugs using GA optimization.	52
4.20	Distribution of log ₁₀ IC ₅₀ values in discovered cancer drugs using GA optimization compared to clinically approved cancer drugs.	56

4.21 Distribution of $\log_{10}IC_{50}$ of clinically approved cancer drugs and <i>de novo</i> compounds proposed by GA optimization using SELFIES IC_{50} shaped latent embeddings.	60
A.1 Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA DeepSMILES latent representations.	67
A.2 Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA SELFIES latent representations.	68
A.3 Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit SMILES latent representations.	69
A.4 Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit DeepSMILES latent representations.	70
A.5 Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit SELFIES latent representations.	71

1

Introduction

1.1 Motivation

Every nine years since the 1950s, the United States Food and Drug Administration (FDA) has approved half as many drugs per billion USD invested, than in the previous nine years. In the last few years, fewer than 0.01% of proposed candidate drugs have obtained market approval (with often a 10-15 year market release cycle) [56]. Discovering and screening new drugs has thus become extremely difficult. Compounding the difficulty of obtaining regulatory approval, the chemical space of bioactive drug-like compounds contains approximately $10^{30} - 10^{60}$ different molecules [48]; finding and screening new drugs by enumeration is computationally impossible. Finding new compounds with high efficacy and low toxicity has thus become a key bottleneck in drug research and development [67]. One of most expensive and time-consuming processes, in particular, is the discovery of new anticancer drugs. Over 34% of drug companies' investment funds go toward the development of anticancer compounds [38]. Thus, being able to design and create anticancer drugs more effectively could both speed up research, save billions of dollars, and save lives.

At the same times as this slowdown in drug research efficiency, the field of machine learning has blossomed over the past ten years. New approaches have enabled significant advances into how to better design compounds tailored to specific diseases. Geeleher

et al. [21] showed that by taking transcriptomic data (DNA expression), researchers could accurately predict cells’ sensitivity to particular drugs. Gómez-Bombarelli *et al.* further demonstrated that compounds could be represented continuously in a latent space and optimized for specific properties using machine learning approaches [24]. Mancia *et al.* [41] further illustrated that machine learning could be utilized to predict a drug’s efficacy against specific genomic profiles by using a subset of genes. Finally, Pu *et al.* [49] showed the public datasets of available drugs can be used to estimate a drug’s probability of being toxic. These approaches have bolstered the promise of personalized drug therapy [41], faster drug discovery, and more efficient drug screening. They have illustrated that by combining information about the molecular structure of compounds, the genetic information and data about drug responses, novel compounds and even optimized drugs can be automatically designed and screened for toxicity, reducing one of the major bottlenecks to drug development.

Given these novel approaches to drug discovery, this work focuses on developing a novel approach to designing new anticancer compounds, optimized for high efficacy and low toxicity, and targeted at specific genomics profiles. By utilizing the latest developments in machine learning, we show how a variety of these methodologies can automate discovery of drugs, personalized medicine, and speed up the drug discovery pipeline.

1.2 Related Work

The closest analogue of our work is *Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules* by Gómez-Bombarelli *et al.* [24]. Gómez-Bombarelli *et al.* utilized a variational autoencoder to convert discrete representations of compounds to and from a continuous latent space. By training an auxiliary network to predict desirable properties like aqueous solubility during training, they managed to shape the continuous representation of molecules effectively to have gradients in directions of desirable properties (*i.e.* drugs with a given property were all in one part of the latent space, while those without it were in another). Gómez-Bombarelli *et al.* then optimized specific desirable drug-like properties in order to generate promising drug candidates. Grechishnikova *et al.* [25] used an alternative approach based on machine translation.

They attempted to decode specific drug compounds by performing translation from the amino acids of target proteins to specific drugs. (Amino acids are the building blocks of proteins, which together with peptides, DNA, RNA, lipids, sugars, metabolites, and other molecules make up the cell [45].) Aumentado-Armstrong [3] optimized the latent space of a variational autoencoder to generate molecules with high binding affinity to target specific protein sites. Mendez-Lucio *et al.* [43] used a generative adversarial network (GAN) to design novel compounds against desired targets, represented by their gene expression signatures. We note here (because their approach is similar to ours) as does Born *et al.* [10], there exists a major issue with this approach. Over 97% of anticancer drugs fail in clinical trials. Often the mechanism suspected for providing a drug’s efficacy is incorrectly identified. Lin *et al.* [36] further showed that in ten drugs in ongoing trials, all of the proposed modes of action were inaccurate. This undermines the approach of using target identification for the discovery of potential drug candidates. Finally, Born *et al.* [10] used reinforcement learning and transcriptomic data to optimize and bias compound generation from a variational autoencoder to be effective against individual cell lines.

1.3 Contribution

In this work we seek to generate novel anti-cancer drug candidates, *in silico*, aimed at specific cancer cell targets and then optimize them for high efficacy and screen them for low toxicity, also *in silico*. To do this, we utilize a variational autoencoder to create highly meaningful representations of chemical compounds in a continuous low-dimensional space. From these representations we predict several desirable properties including QED (quantitative estimate of drug-likeness); solubility in water ($\log P$, or the base-ten logarithm of the water-octanol partition coefficient); and solvent accessible surface area (SAS). In addition to calculating these important drug properties, we then seek to shape the latent space of our variational autoencoders to be responsive to the transcriptomic profile of given cell lines (*i.e.* compounds with low efficacy are in one area of the latent space, while those with high efficacy are in another). Using this shaped and highly information-dense latent representation, we seek to predict compounds’ toxicity and efficacy (as IC_{50}) on an individual cell line conditioned on its transcriptomic profiles.

Finally, we seek to optimize compounds' efficacy against individual cancer cell lines and against particular types of cancer using a genetic algorithm. This work is sectioned then into five main algorithmic components.

First, we train a variational autoencoder to take discrete chemical representations of drug compounds to and from a latent space. Our variational autoencoder further uses several techniques to ensure the high information density of the latent space. We utilize specialized annealing schedules and implicit latent codes. Recent works have shown these techniques to vastly improve the information density of latent spaces. We furthermore seek to shape the latent space of our variational autoencoder to create a gradient for the efficacy of chemical compounds against particular cell lines. This allows us to predict efficacy values more effectively downstream from the latent space and to optimize our proposed novel chemical compounds more efficiently. As far as we know, we are the first to shape the latent space of a variational autoencoder specifically to have a gradient for the efficacy of chemical compounds against particular cell-lines.

Second, we train a specialized transformer decoder to take pre-trained embeddings of compounds from the latent space back to their corresponding discrete chemical representations. This approach enables more locations of the continuous latent space to be accurately decoded back to discrete representations. As far as we are aware, we are the first to do this within this context.

Third, we develop a novel architecture to predict the efficacy of proposed compounds against particular cell lines. We take a compound's latent space representation, which is informationally dense and the transcriptomic data of cell lines, outputting the efficacy of the proposed drug.

Fourth, we develop a means of predicting the toxicity of compounds from their latent space representation using an ensemble of different machine learning algorithms including random forests, extra-trees, and quadratic discriminant analysis.

Fifth, we use both Bayesian optimization and a genetic algorithm to optimize proposed anticancer compounds. We show that by using our predictor of a candidate compound's efficacy against a particular cell line, we can move within the latent space to make it more potent. Utilizing our more powerful decoder, which is able to decode more

places within the latent space, we can finally obtain highly potent and specialized novel compounds. We then apply this same approach to find new compounds that are designed to be effective against particular types of cancer/histologies and cancer sites (*i.e.* lung, breast etc). We finally rank our novel compounds using their efficacy against particular cell lines and against specific histologies.

We show in this work the high probability of personalized medicine and how machine learning and novel statistical techniques can be used efficiently to discover and screen new drug candidates. Our work can be used for a variety of purposes most notably specialized medicine. Because we utilize the genetic information of particular patient's cells, novel proposed drugs can be developed specifically for given patients. Most importantly our approach can be used to generate a plethora of novel drug candidates by aggregating large batches of patient genetic data. New drugs can be designed to target specific types of cancer and screened efficiently and easily using our method.

2

Background

2.1 Introduction

First, we give background on drug discovery. We further discuss several recent developments in the field of *in silico* (by use of computer) drug discovery. We then showcase machine learning frameworks that have been developed to assist *in silico* drug discovery.

Second, we give an overview of the main machine learning technique that we utilized in this work: variational autoencoders. We elaborate on the details on how we solved a major issue for variational autoencoders: *posterior collapse*.

Last, we give a brief overview of several machine learning algorithms that we used to predict various quantities and characteristics of chemical compounds.

2.2 Drug Discovery

Eroom’s Law states that, since the 1950s, the productivity of the drug discovery pipeline (measured by the number of FDA approved drugs per invested USD) halves every nine years [57]. Less than 0.01% of synthesized drug candidates obtain market approval and these compounds often have a market release schedule of 10-15 years. Costs have risen in tandem with this productivity slowdown, ranging from one to three billion USD per drug [56].

Decision Point	Criteria for Decision	Evidence Used
<i>Choice of Disease</i>	Patient Need; Commercial Aspects	Statistics on disease distribution
<i>Target Selection</i>	"Validated target" (<i>i.e.</i> druggable)	Biological studies
<i>Screen library assembly</i>	Chemistry with no liabilities, ease of synthesis	Cheminformatics analysis of chemical space
<i>Assay development</i>	Predictivity; reproducibility; price	Experience of biologists
<i>Screen/hit list triaging</i>	Data quality; certainty of TPR and FPR	Experience of screeners
<i>Lead Optimization</i>	On-target and Off-target activities; pharmacological properties	Gene-expression arrays; more complex assay systems
<i>Pre-clinical studies</i>	Efficacy and side-effects	Animal experiments
<i>Clinical studies</i>	Efficacy and side-effects	Large human testing
<i>Approval</i>	Efficacy and side-effects	Results from pre-clinical and clinical studies

Table 2.1: Some of the typical decision points in a drug discovery project. This chart illustrates the many points of failure in drug discovery [65]. This work focuses on the *lead optimization stage*.

As seen in Table 2.1, there are a many stages at which a promising drug candidate can fail. The lower efficiency of drug discovery though has been attributed to the high dropout rate of candidate compounds in some of the early stages of the pipeline. Drug candidates must be properly screened in earlier phases; therefore ADMET (absorption, distribution, metabolism, excretion, toxicity) assessments and high-throughput screenings are essential. Already 97% of anticancer candidate drugs fail in clinical trials in the USA and never receive approval, which further emphasizes the need for more intensive screening. The costs of the latter clinical phases are simply overwhelmingly prohibitive and any tougher pre-screening approach that reduces these failures' number is imperative.

A major issue in pre-screening is the exploration of chemical space which is estimated to contain $\sim 10^{30} - 10^{60}$ drug-like molecules with bio-active properties. Finding a method of effectively exploring this chemical space and pre-screening the molecules has thus occupied a great deal of research.

2.2.1 Desirable Drug Properties

We present background on four different criteria used to determine the promise of a drug: **logP**, **QED**, **SA score**, and **IC₅₀**.

Water-octanol partition coefficient: logP

People often prefer to take a pill rather than have an injection. The partition coefficient (P) gives the propensity of a 'neutral compound to dissolve in an immiscible biophasic system of lipid (fats, oils, organic solvents) and water [7]. Accordingly, it measures how much of the compound dissolves in the water portion versus the organic portion of the solution. A negative value means that a compound is hydrophilic (dissolves more in water). A positive value indicates that the compound is lipophilic (dissolves more in the lipids). Using log of this value (logP) as a shorthand, researchers can determine whether a substance can be absorbed by humans or other types of living tissue.

Drug candidates are often screened by their computed logP value. In general, a compound intended for oral administration should have a logP less than 5 [9].

Quantitative Estimation of Drug-likeness: QED

The Quantitative Estimation of Drug-likeness or QED is a score used to evaluate a drug-like compound's favourability. Before the advent of this score, drug-likeness was normally assessed using Lipinski's Rule of Five [9]. Lipinski's Rule of Five states that a compound is unlikely to exhibit desirable absorption or permeation when two of or more of the following are fulfilled:

1. the molecular weight is greater than 500 Da (Daltons).
2. the calculated logP is greater than 5.
3. there are more than 5 hydrogen-bond donors or the number of hydrogen-bond acceptors (N and O atoms) is greater than 10.

However, in the past, Lipinski's rule caused researchers to filter out promising candidates and sometimes future treatments. In response QED was proposed as a quantitative measure of several multidimensional criteria to assess a compound's drug-likeness. QED values range from 0 (all properties unfavourable) to 1 (all properties favourable) [9]. For more information on its exact calculation see [9].

Synthetic Accessibility Score: SA Scores

Synthetic Accessibility Score measures the ease of synthesizing a particular compound. Developed by Ertl and Schuffenhauer, SAS combines information about the chemical fragments within individual compounds and experimental data of synthesized compounds [16].

In order to build the criteria for SA scores, a subset of 1 million compound structures from the online database PubChem [32] were fragmented into their components. These fragments were then scored according to their frequency of occurrences, with the most frequently occurring receiving the highest scores. In addition to these fragment counts, other factors such as ring complexity are considered in formulation of the score [16].

SA scores are on a scale of 1 to 10, with 1 being readily synthesized and 10 being nearly unsynthesizable [16].

Half-Maximal Inhibitory Concentration: IC₅₀

The half-maximal inhibitory concentration or IC₅₀ is a measure of a drug's efficacy. It measures how much of a drug is needed to inhibit a biological process by 50%. IC₅₀ is thus an excellent measurement of a biological process's sensitivity to a given compound. For cancer research especially, IC₅₀ values provide a valuable way of examining a drug's potency [4]. IC₅₀ are given in molars (M). Potent inhibitors are those with IC₅₀ values on the order of 10⁻⁹ M (*i.e.* nM). Often IC₅₀ values are converted to the negative log₁₀ scale: (pIC₅₀) for simplicity. However, in this work we use log₁₀ IC₅₀ in order to be consistent with previous works that predict IC₅₀ [41, 13].

Tanimoto Similarities

Separate from the above four drug properties, chemists often want to know the similarity between two compounds. Chemists and researchers often use a similarity measure called the Tanimoto similarity between the chemical fingerprints of two molecules [5]. Fingerprints are an abstract way of representing certain chemical structural features of molecules [14, 55]. Researchers generally consider two structures to be similar if their Tanimoto similarity is greater than 0.85 [14]. Tanimoto similarities generally use Daylight

fingerprints (fingerprinting techniques developed by Daylight Chemical Information Systems Inc. [14]) to calculate Tanimoto similarities.

In addition to the above four characteristics and Tanimoto similarities, researchers also must understand biological environments into which drugs are placed. One of the most important aspects of these environments in humans is gene expression. A measure of this expression is given by transcriptomics.

2.2.2 Transcriptomics

The human genome is made up of deoxyribonucleic acid (DNA), long double-helical molecules that contain information about how to create and maintain the functionality and structure of cells [63]. Specifically, the instructions within DNA consist of four different chemicals: cytosine (C), guanine (G), adenine (A), and thymine (T). These four chemicals are repeated in different orders and lengths creating 20,000 to 25,000 different genes in humans.

However, DNA must be read and transcribed into ribonucleic acid (RNA) for its instructions to be carried out. The transcribed gene readouts or RNA are called transcripts and the transcriptome is the collection of all the RNA in a cell [63]. A particularly important type of RNA is messenger RNA (mRNA). mRNA transcripts are delivered to ribosomes (molecular structures in a cell) that then translate the mRNA in order to assemble building blocks called amino acids into structures called proteins.

The transcriptome mirrors the sequences of DNA from which it was transcribed. The level or amount of the RNA transcribed indicates a particular gene's activity- also known as its gene expression [63]. By analyzing the transcriptome of a cell, researchers can thus determine whether a particular gene is turned off or on and the particulars of the cellular environment.

Transcript profiling or transcriptomics is a technique that obtains information on the frequencies of particular mRNA transcripts in cells [23]. The two main techniques used in transcriptomics are *microarrays*, which quantify a set of predetermined sequences, and *RNA-sequencing*, which uses *high throughput sequences* to capture all sequences [39].

See [39] for more details about transcript profiling techniques. Our work focuses on transcriptomic data collected through RNA-sequencing.

Increasingly researchers have used transcriptomic data for characterizing the biological effects of small molecule drug candidates. De Wolf *et al.* [15] in particular successfully showed that the transcriptomic profile, acting as a metabolic signature, can be used for *de novo* drug identification. Born *et al.* [10] showed that through reinforcement learning, drug generation in the disease context represented by a transcriptomic profile can produce higher efficacy drug candidates. Transcriptomics have thus been used extensively to screen drug candidates. Like Born *et al.* [10], we will propose a framework to generate novel drug candidates based solely on a tumour's metabolic signature (as opposed to targeting or incorporating information about potential targets into the design process), represented by the transcriptome of a cell.

In order to represent and generate drug candidates, we required a means of digitally specifying their chemical structure. We turned to SMILES (and its variants for this purpose).

2.2.3 Simplified Molecular Input Entry System (SMILES)

SMILES [58] is a line notation for representing molecules and reactions. For example, "CC" represents the compound ethane.

Chemical	SMILES	DeepSMILES	SELFIES
Ethane	CC	CC	[C][C]
Carbon Dioxide	O=C=O	O=C=O	[O][=C][=O]
Benzene	c1ccccc1	cccccc6	[c][c][c][c][c][c][Ring1][Branch1 ₁]
Acetic Acid	CC(=O)O	CC=O)O	[C][C][Branch1 ₃][epsilon][=O][O]

SMILES are fairly compact compared to other ways of representing a compound's structure. Despite their relative ease of use, generative machine learning models often have difficulty with them. The generated SMILES strings can often be syntactically invalid. As a result of these issues, two additional methods of representing chemicals for the purpose of machine learning were developed.

2.2.4 DeepSMILES

DeepSMILES [46] are an alternative line notation for representing molecules built for machine learning. DeepSMILES specifically address two issues that SMILES strings encounter in generative models. First, DeepSMILES syntax avoids the problem of unbalanced parentheses by only using close parentheses. Secondly, DeepSMILES avoids the issue of pairing ring closure symbols by only using a single symbol for ring closures. DeepSMILES strings can be directly translated back and forth to their corresponding SMILES strings.

2.2.5 Self-Referencing Embedded Strings (SELFIES)

SELFIES [34] are alternative string-based representations of molecular graphs that are 100% robust. Namely, each SELFIES string corresponds to a valid molecule. (We note that while all SELFIES correspond to a *valid* molecule, many of these *valid* molecules are not chemically viable [34]. SELFIES encode branch length as well as ring size in corresponding *Branch* and *Ring* identifiers, allowing for 100% robust representation.

2.2.6 SPVec

Although we do not use it within our work, SPVec [70] is a Word2Vec [44] inspired manner of representing SMILES in a continuous space. Based on a Skip-gram model implemented with negative-sampling [70], SPVec manages to project compounds to a low dimensional space where biophysical and biochemical properties can be easily predicted. We leave incorporating SPVec into this project as future work.

After representing drug-like candidates using SMILES and its variants, we needed a means of representing molecules in a continuous space in order to optimize them for drug discovery. Variational autoencoders, a machine learning technique, are a known algorithm of reducing the dimensionality of data to a smaller continuous space.

2.3 Variational Autoencoders

Neural Networks can define a probabilistic model for nonlinear dimensionality reduction. One of the most popular and useful examples of these *deep generative models* is the variational autoencoder (VAE).

A variational autoencoder model encodes points, X , with dimensionality p into latent variables, Z , with dimensionality $k < p$, with Z having a given prior $p(z)$. This joint probability can be written as $p(X, Z) = p(X|Z)p(Z)$. With this formulation, new points, X_i , can be generated in the following way:

1. Draw a latent variable $Z_i \sim p(Z)$.
2. Draw a datapoint $X_i \sim p(X|Z)$.

The goal then of the variational autoencoder is perform posterior inference over the latent variables given observed data to recover good approximations of points simulated as latent variables:

$$p(Z|X) = \frac{p(Z|X)p(Z)}{p(X)} \quad (2.1)$$

Unfortunately, performing this calculation is intractable. As a result, variational autoencoders approximate the posterior $p(Z|X)$ using a neural network. This work is done by an *encoder* neural network. The encoder, a neural network parameterized by weights θ takes input points datapoint, X_i , outputting their latent representation, Z_i . Each latent variable Z_i has a standard Gaussian prior:

$$Z_i \sim N(0, I_k)$$

The encoder is thus denoted as $q_\theta(Z|X)$. Using the θ -parameterized encoder's variational distribution $q_\theta(Z|X)$, the variational autoencoder thus approximates $p(Z|X) \propto p(X|Z)p(Z)$.

The *decoder*, another neural net, performs the second half of calculations for a VAE. Parameterized by a different set of weights ϕ , the decoder takes in a latent variable, Z_i ,

outputting a reconstruction, \tilde{X}_i , of X_i . The decoder is thus denoted as $p_\phi(\tilde{X}|Z)$. The key insight is that the generative decoder can be nonlinear and non-Gaussian.

$$\tilde{X}_i|Z_i \sim p_\phi(\cdot|Z_i)$$

The decoder thus tries to maximize the marginal data log-likelihood, $\mathbb{E}_{X \sim D}[\log p_\phi(X)]$, minimizing the reconstruction error.

The VAE thus must learn to encode and decode points to and from a latent space. However, while training the VAE, its training algorithm must ensure that the encoder properly approximates the true posterior, $p(z|x)$. To do so, the VAE algorithm utilizes the Kullback-Leibler divergence $\text{KL}(q_\theta(Z|X)||p(Z|X))$, which measures the information loss when using q to approximate p . As noted before, this is intractable, so this value is approximated using the evidence lower bound/variational free energy (ELBO). The ELBO in a latent variable model $p(X,Z|\theta)$ is defined as:

$$\mathbb{E}_{x \sim D} \log p(\mathbf{x}) \geq \mathcal{L}_1 = \mathcal{L}_2 \quad (2.2)$$

with:

$$\mathcal{L}_1 = \mathbb{E}_{x \sim D} [\mathbb{E}_{z \sim q_\theta(z|x)} \log p(\mathbf{x})] + \mathbb{E}_{x \sim D} [-\text{KL}(q_\theta(z|x)||p(z))] \quad (2.3)$$

and:

$$\mathcal{L}_2 = \mathbb{E}_{x \sim D} \log p(\mathbf{x}) - \mathbb{E}_{x \sim D} \text{KL}(q_\theta(z|x)||p(z|x)) \quad (2.4)$$

\mathcal{L}_1 consists of the reconstruction error term and a KL divergence regularization term. The first term is the expectation of reconstruction loss (log-likelihood) over the encoder's distribution of latent representations Z . The Kullback-Leibler (KL) divergence regularization term is between the encoder output, $q_\theta(Z|X)$, and the prior, $p(Z)$. As noted, the prior, $p(Z)$, is ordinarily chosen as a standard Gaussian. This term penalizes the encoder output if the latent representation, Z , is significantly different from the standard Gaussian distributions. This KL term causes similar inputs, X , to have similar representations, Z , in the latent space. This effectively balanced the goals of the encoder and decoder.

The alternative \mathcal{L}_2 formulation indicates that the variational autoencoder requires a flexible encoding distribution family to minimize the approximation divergence between the true posterior and the best encoding distribution.

To train the VAE, a stochastic algorithm thus optimizes this ELBO.

2.3.1 Posterior Collapse

With the \mathcal{L}_1 formulation, the VAE objective consists of two terms (1) reconstruction and (2) KL regularization. These terms are often balanced by a weighting hyper-parameter, β . However, during training, the KL-term often tends to vanish as the latent space comes to match its prior.

Posterior collapse occurs when the latent space variational distribution nearly matches its prior [40]. As a result of this phenomenon, the variational autoencoder generative model learns to disregard a subset of the latent variables. The model essentially learns to predict and "know" what the other latent dimensions are from one or two dimensions of the latent representation. The latent variables thus become uninformative. Posterior collapse reduces the capacity of the generative model, in effect making it impossible for the decoder to utilize the content of the latent dimensions. Posterior collapse is caused by the KL-divergence term in the ELBO objective because it actively encourages the variational term to move towards zero, making the latent codes match the prior. Furthermore, powerful decoders can often disregard sections of the latent space in order to decode the variables, further encouraging the latent variables to match the prior. In our work, this causes latent codes to match a Gaussian prior:

$$\exists i \text{ s.t. } \forall x q_{\theta}(Z_i | X_i) \approx N(0, I_k)$$

Using the true posterior $p(Z|X)$, this can be written as:

$$\exists i \text{ s.t. } \forall x p(Z_i | X_i) \approx N(0, I_k)$$

2.3.2 Cyclical KL Annealing

Posterior collapse can prevent the VAE from learning informative latent codes. For this reason, slowly increasing the β hyper-parameter during training is often used. This approach is known as *annealing*. However, if trained for multiple epochs, the KL-term can often still disappear.

In 2019, Fu *et al.* [20] proposed a cyclical annealing schedule, which repeats the process of increasing β multiple times. "This process encourages the progressive learning

of more meaningful latent codes, by leveraging the informative representations of previous cycles" [20]. This approach was validated on a broad range of natural language processing (NLP) tasks, illustrating its usefulness.

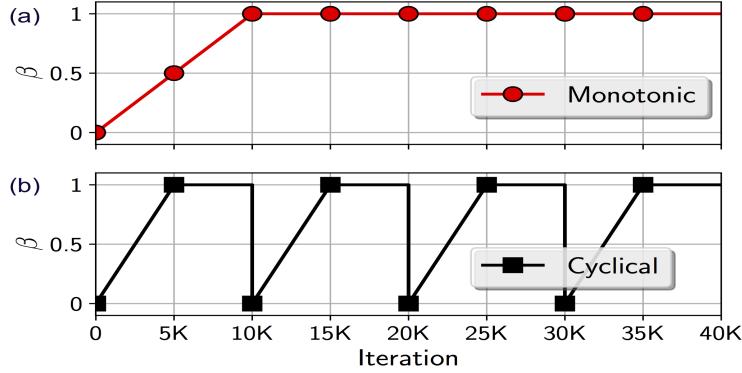


Figure 2.1: KL Regularization term in Cyclical Annealing. Figure from [40].

2.3.3 Implicit Latent Codes

The implicit deep latent variable model is another approach to posterior collapse advocated by Fang *et al.*[17]. This approach seeks to create smooth latent structures to guide generation and to discard the limitation of utilizing a Gaussian distribution for the prior of the latent space. This approach specifically uses a sample-based representation of variational distributions to create *implicit* latent features, which provide flexible representations compared to Gaussian-based posteriors.

Implicit Representations

Instead of assuming an explicit prior over the latent distribution, $q_\theta(Z|X)$, a sampling mechanism represents a set of samples, $\{Z_{X,i}\}_{i=1}^M$, through the encoder as:

$$Z_{x,i} = G_\phi(x, \varepsilon_i), \varepsilon_i \sim q(\varepsilon) \quad (2.5)$$

where the i -th sample is drawn from a separate neural network, G_ϕ , that takes (x, ε) as input; ε_i is a sample from $q(\varepsilon)$, and $q(\varepsilon)$ is a simple standard Gaussian. To combine the noise with the input sequence, x , ε_i is appended with a hidden representation of the sequence, x , generated by the encoder. Fig. 2.2 illustrates this.

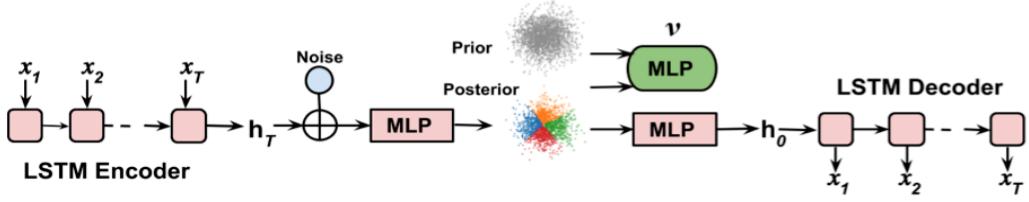


Figure 2.2: Diagram illustrating the network structure of the implicit latent model VAE. Figure from [17]. X_i are time-series inputs at time i , h_i are hidden states computed by the model for time step i . For details on multi-layer perceptrons (MLP), Long short-term memory (LSTM) see Section 2.4.

Dual form of KL-Divergence for Implicit Representations

The implicit representations defined by Equation 2.5 cause issues during training as the KL term is no longer tractable. However, by optimizing the KL divergence dual form based on the Fenchel duality theorem [54], this issue can be circumvented:

$$\text{KL}(q_\theta(z|x) || p(z)) = \max_v \mathbb{E}_{Z \sim q_{\text{theta}}(Z|X)} v_\psi(X, Z) - \mathbb{E}_{Z \sim p(Z)} \exp(v_\psi(X, Z)) \quad (2.6)$$

where $v_\psi(X, Z)$ is an auxiliary dual function parameterized by a neural network with weights ψ .

After replacing the KL term with the dual form, the implicit variational autoencoder has the following objective:

$$\mathbb{E}_{X \sim D} [\mathbb{E}_{\text{sim}q_\theta(Z|X)} \log p(\mathbf{X})] - \mathbb{E}_{X \sim D} [\mathbb{E}_{Z \sim q_\theta(Z|X)} v_\psi(X, Z)] + \mathbb{E}_{X \sim D} [\mathbb{E}_{Z \sim q_\theta(Z|X)} v_\psi(X, Z)] \quad (2.7)$$

As noted in Section 2.3.1, the main issue with VAEs is posterior collapse. To better regularize the space, Fang *et al.* further proposed replacing the KL-divergence term, $\mathbb{E}_{X \sim D} - \text{KL}(q_\theta(Z|X) || p(Z))$ with:

$$-\text{KL}(q_\theta(Z) || p(Z)) \quad (2.8)$$

where $q_\theta(Z) = \int q(X)q_\theta(Z|X)$ is the aggregated posterior and $q(X)$ is the empirical data distribution of the training dataset. The integral can be estimated by first sampling x and then sampling $Z_i \sim q_\theta(Z_i|X_i)$. By using Equation 2.8, the variational autoencoder works to represent each point, X_i , as a local point in the latent space such that the aggregated latent points, Z , match the prior.

This approach coincides with Zhao *et al.*'s [71] approach where mutual information is introduced into the optimization process. We do not present the proof of their result here, showing only the useful (relative to our work) aspects of their work. Based on their decomposition result we see:

$$\text{KL}(q_\theta(z)||p(z)) = I(x,z) - \mathbb{E}_X \text{KL}(q_\theta(z|x)||p(z)) \quad (2.9)$$

where $I(x,z)$ is the mutual information between Z and X under the joint distribution $q_\theta(X,Z) = q(X)q_\theta(Z|X)$. The objective in Equation 2.7 thus also maximizes the mutual information between data points X_i and their latent features Z_i .

2.4 Additional Machine Learning Algorithms

We use a host of other machine learning methods within this work to predict toxicity, to better control the latent space, and to optimize the properties of newly found chemical compounds.

2.4.1 Machine Learning for Manipulating the VAE Latent Space

We first present several machine learning algorithms that we used to map discrete representations of chemical compounds back and forth from a continuous latent space.

Neural Networks: Multi-layer Perceptrons

Multi-layer perceptrons (MLP) are a class of feed-forward neural networks composed of variously connected layers of artificial neurons that are activated upon reaching a threshold. Each hidden layer within an MLP is composed of multiple nodes that consist of a linear function composed with a nonlinear activation function [66].

$$f(x) = \sigma(W \cdot x + b)$$

where $\sigma(x)$ is a nonlinear activation function (*i.e.* $\text{ReLU}(x) = \max(0,x)$ or $\tanh(x)$), W is a matrix of weights, and b is an added bias. The final layer of MLPs used for classification often output a set of probabilities for each class by using a *Softmax* layer:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

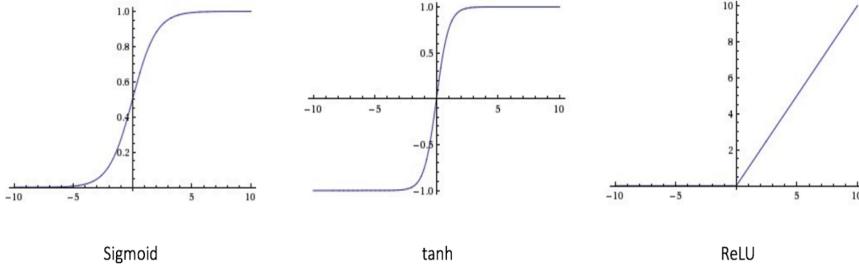


Figure 2.3: Sigmoid, Tanh, and ReLU. Figure from [59]

Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network particularly adept at extracting 'strong signals' from data [8]. CNNs utilize layers with convolving filters that are applied to local features and thus extract more complicated features from these local features. Convolutional layers accomplish this by making use of an input translation invariance (*i.e.* features appearing in multiple places). This allows locally connected features to convolve over the entire feature map.

Pooling layers are often used after convolving layers within CNNs. Pooling layers combine adjacent outputs from feature maps and output the maximum. In this way, pooling layers down-sample convolutional layers in order to extract the strongest signals.

Finally, fully-connected layers at the end of networks are dense layers that have every output neuron of the penultimate layer as input to each neuron in the last layer.

Long-Short Term Memory (LSTM) Recurrent Neural Networks

LSTMs are a recurrent neural network (RNN) that make use of feedback connections [42]. Each sequential element an LSTM consists of a cell. Each of these cells can erase, write, and read information. LSTMs can thus learn the context of a given character or word in a sequential/time-based input by learning long-term information.

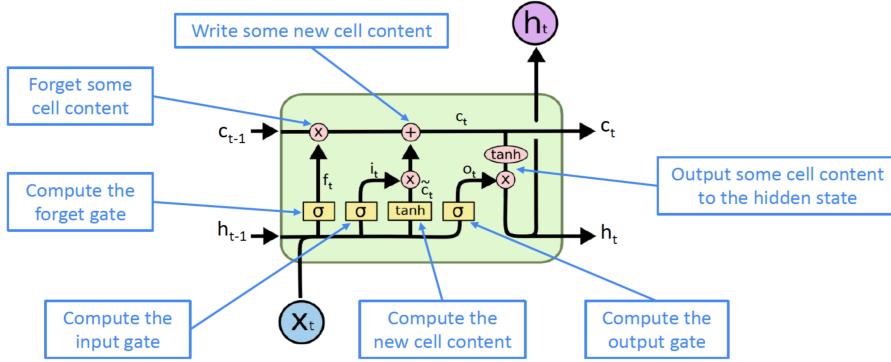


Figure 2.4: LSTM Cell. Figure from [42].

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_C)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

where the σ s represents sigmoid operations, W s are matrix of weights, b s are added biases, h_t is the hidden state computed at each time step t ., C_t is the cell state computed at each time-step t ., and x_t is the series input at time-step t [42].

We used LSTMs in order to learn features of the chemical compounds in our model from their discrete representations.

Transformer

Powerful decoders are rarely used in variational autoencoders in order to prevent posterior collapse (see Section 2.3.1). For this reason, after training various VAES and obtaining information-dense embeddings, we trained a more powerful transformer to more flexibly decode the pre-trained latent space autoregressively as in [30]. We give a brief overview of transformers here.

Proposed by Vaswani *et al.* [64], the transformer is a sequence-to-sequence machine learning architecture that does not rely on RNNs. Transformers most importantly make use of self-attention.

Self-Attention: Language often relies heavily on context. Self-attention attempts to understand the relevant associated words/characters when processing a given word/character.

For example, in the sentence "Statistics is one of the most important subjects in the world!", a transformer would attempt to utilize the word "Statistics" when trying to understand the context of "subjects". Self-attention specifically works by processing each word vector in a given segment by making use of three novel components:

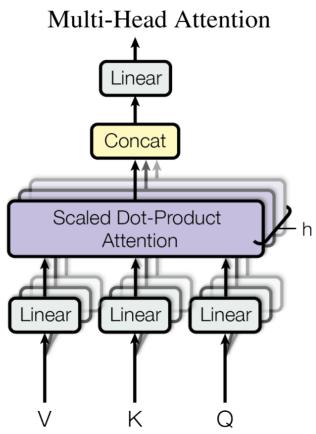
1. Query: The query is a representation of the current word/character that is scored against all other words/characters.
2. Key: Key vectors are labels for all the words/characters being considered.
3. Value: Value vectors are character/word representations. Values are sums of each word/character's relevancy score.

An attention function maps queries and key-value pairs to an output. A scaled dot-product then calculates the correlation between the query Q and the key K , multiplying by V [64], giving:

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{Q * K^T}{\sqrt{d_k}}\right)V \quad (2.10)$$

where d_k is the dimension of the key space.

Transformers also use multi-head attention.



Multi-headed attention allows the model to attend to information at h positions of the sentence simultaneously.

$$\text{Multi}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^K)$$

and the matrices W^O , W_i^Q , W_i^K , and W_i^K correspond to linear projections.

Transformers normally consist of an encoder-

Figure 2.5: Multi-headed Attention. Figure from [64].

decoder structure. We only make use of the decoder aspect in our transformer. For this reason, we do

not discuss the encoder architecture. See [64] for more details. Like in Radford *et al.*'s GPT-2 Transformer [50], we jettison the encoder-decoder attention layers in our decoder. The remaining architecture consists of several decoding layers, each consisting of a multi-headed attention layer, and a feed forward layer.

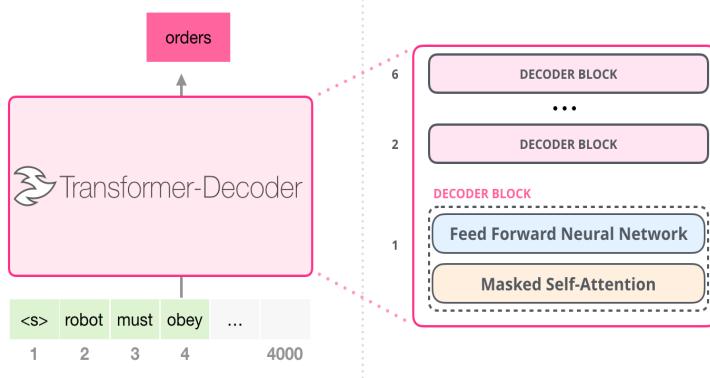


Figure 2.6: Decoder-Layer and resulting full Decoder. Figure from [1].

2.4.2 Machine Learning Algorithms for Predicting Chemical Toxicity

We now present background for several machine learning methods we utilized to estimate the toxicity of various proposed chemical compounds.

Random-Forests

Random forests are an ensemble of decision trees [11]. Specifically, each tree predictor performs a series of decisions that are used to conduct finer and finer grained analysis (*e.g.* "Is this object a fruit?", "Is this fruit round?", "Is this bigger than an orange"?). Each tree is initialized from an independently sampled randomized vector [11]. Random trees, importantly, make use of bootstrap samples of the training dataset, further allowing individual trees to learn different information. When performing splits on individual trees, random forests utilize a greedy algorithm to decide which feature to split.

Random forests themselves consist of many individual decision trees that collectively make a decision. Random forests can be used for both regression and classification. For classification for example, each decision tree could report a class prediction with the winning class having the plurality of votes. For regression, the output could be a weighted average of the decision trees. Each of these trees, both in regression and classification, are initialized differently and thus act independently.

Extremely Randomized Trees/Extra-Trees

Extremely Randomized Trees or Extra-trees are a similar ensemble algorithm to random trees. Extra-trees however train on the entire training set (no bootstrapping) when constructing individual trees. In addition, unlike random trees, extra-trees do not perform tree-splits on randomized features at each level [22].

Dropouts Meet Multiple Additive Regression Trees (DART)

DART is a gradient boosting tree ensemble algorithm [51]. This algorithm uses dropout [60] on boosted regression trees to make accurate yet generalizable tree models.

DART, at each training iteration, computes the derivative of the loss function and adds a tree that fits the inverse of this derivative to the ensemble. DART also during training performs dropout of previously trained trees in its ensemble when fitting new trees. In this way DART attempts to prevent overfitting [51].

Support Vector Machines

Support vector machines (SVM) are a machine learning algorithm used for classification [61]. SVMs map input points to a hyperspace in order to separate input classes as far as possible. SVMs then find the hyperplane that maximizes the margin, (twice the smallest distance from each class to the separating hyperplane).

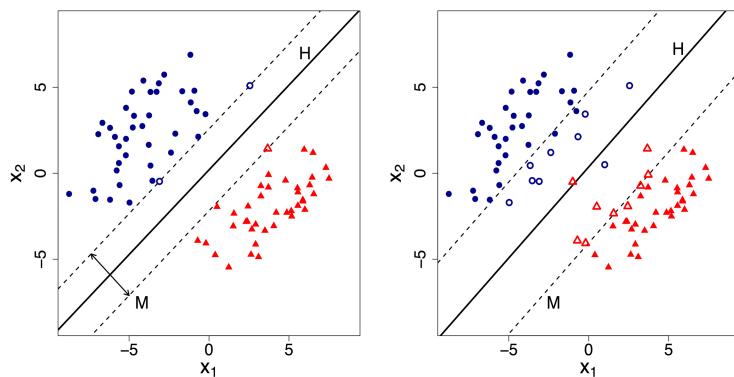


Figure 2.7: SVM with dividing hyperplane H and margin M for a two-dimensional dataset (x_1, x_2) . Figure from [33]. The left shows a completely separable dataset while the right shows an inseparable set.

SVMs are an inordinately useful machine learning algorithm and are able to provide the probability of classification of each point by using the distance of the point to the separating hyperplane [61].

2.4.3 Optimization Algorithms

In order to optimize compounds within the latent spaces for certain desirable properties, various approaches have been proposed including unconstrained and constrained Bayesian optimization [24, 27]. We now present the algorithms that we used to optimize several different desirable properties of proposed anticancer chemicals compounds.

Genetic Algorithms

We propose utilizing a genetic algorithm for discovering new compounds with desirable properties. Genetic algorithms are an evolutionary computational approach. In this approach, a randomly generated population of individuals is used to generate a new population through variation operations. Variations often include operations like mutation and crossover. By then selecting the **best** individuals according to a fitness function, the genetic algorithm optimizes said fitness function. In our case, we plan to apply this approach by using the latent space's dimensions as "genes" that define each "individual" chemical compound.

Algorithm 1 Genetic Algorithm [18]

```
1: procedure GENETICALGORITHM(GENERATIONS, POPSIZE, PROBLEMARGS)
2:   population = RandomPopulation(popsze)
3:   for i=0 to generations do
4:     population = VariationOperations(popsze, problemargs)
5:     population = problemargs.FitnessFunc(population)
6:     population = problemargs.sort(objective)
7:     population = BEST(population, popsze)
8:   end for
9:   return BEST(population)
10: end procedure
```

Arithmetic Crossover: Arithmetic crossover occurs when two individuals from the populations are combined using some arithmetic operation. In our work, arithmetic

crossover takes two points in the latent space and combines them using a weighted sum [47].

2.4.4 Bayesian Optimization

Bayesian optimization (BO) is a formalized way to estimate the global optimum of an objective function, $f(\cdot)$, over a bounded domain [19]. Given that evaluations of $f(\cdot)$ are considered expensive, the aim of BO is to keep the number of evaluations small. By making use of non-parametric probabilistic auxiliary models to act as an approximation of the function, $f(\cdot)$, BO can efficiently determine using a different *acquisition function* the next candidate to evaluate with the function, $f(\cdot)$:

$$f(x) : f(x^*) = \max_x f(x) \quad (2.11)$$

where x^* is the optimum point and x are the evaluated points.

The auxiliary function is often modelled using a Gaussian process (GP): $f(x) \sim GP(m(x), k(x_i, x_j))$, as this function is fairly flexible. $k(\cdot, \cdot)$ encodes the similarity between the two points based on a user-selected metric, and the function $m(\cdot)$ give the mean values.

The *acquisition function* is a function utilized to balance between *exploring* other areas of the domain and *exploiting* the current explored areas of the domain. Commonly used acquisition functions include probability of improvement and expected improvement [19].

3

Experimental Setup

3.1 Introduction

We describe the tools and datasets that we used in this chapter. We make use of two datasets to train and evaluate our VAE models: ChEMBL [6] and ZINC [29]. We utilize the GDSC [28] for predicting IC₅₀ data. Lastly to train our models we utilize Python and Google Colab.

3.2 Datasets

The three datasets ChEMBL, ZINC, and GDSC are staples of previous works in chemical compound generation [24, 41, 43, 25, 10]. We utilized these datasets to allow direct comparisons with prior work.

3.2.1 ChEMBL

To model bioactive small molecules for our variational autoencoder, we utilized ChEMBL [6]. ChEMBL is a database with over 1.9 million bioactive molecules with drug-like properties and their associated bioactivities [6]. We constrain our dataset to only contain molecules with fewer than 50 heavy atoms (*i.e.* non-hydrogen) and only composed of H, B, C, N, O, F, Si, P, S, Cl, Br, and I. Our resulting ChEMBL dataset had 1.6 million bioactive compounds.

3.2.2 ZINC

ZINC is another database with bioactive molecules [29]. We used a subset of 250,000 drug-like commercially available compounds from this dataset as in Gómez-Bombarelli *et al.* [24]. We further constrained this dataset in the same way as ChEMBL. We used ZINC to estimate how effectively our model could represent and decode molecules from outside its training dataset distribution.

3.2.3 IC₅₀ Sensitivity: GDSC

We utilized drug sensitivity data from the public database Genomics of Drug Sensitivity in Cancer (GDSC) [28]. "GDSC contains the screening results of over 1000 genetically profiled human pan-cancer cell lines with a wide range of anticancer compounds" [41]. The transcriptomic profiles are all those of cancer cells *untreated*. Drug sensitivity values in this dataset are represented by IC₅₀ values on the **log-scale**. We focus on 202 of the 265 drugs in the dataset that met our criteria of only containing the atoms H, B, C, N, O, F, Si, P, S, Cl, Br, and I. Similarly, we focus on the transcriptomic profiles of the available 985 cell lines. Because the GDSC database lacks certain values, and because of our own criteria, pairing of the 985 cell lines with the anticancer compounds resulted in approximately 151,444 pairs. In training using this dataset, we used randomized SMILES augmentation [2] (different representations of the same SMILES), resulting in more than 4.9 million data points. We used 20% of the cell line pairs for testing and the rest for training.

17,737 genes initially represented each of the 985 cell lines. Manica *et al.* [41] showed this transcriptome could be reduced to a subset of 2128 genes by utilizing network propagation over the STRING protein-protein interaction network [62]. By using STRING, Manica *et al.*'s approach incorporated intracellular interactions by adopting a network propagation scheme for each drug. In the propagation scheme, weights associated with the reported targets were diffused over the STRING network, giving back an importance distribution over the genes. We utilize the 2128 most important genes following Manica *et al.* [41].

3.2.4 Toxicity: DrugBank, KEGG, TOXNET, T3DB

For toxicity prediction, we consider approved drugs as having an appropriate benefit risk ratio and hence to be non-toxic. Though they may still pose a risk, we consider them as non-toxic due to their corresponding clinically approved status and therapeutic properties.

For non-toxic compounds we utilized two datasets: the FDA-approved and the Kyoto Encyclopaedia of Genes and Genomes (KEGG) Drug [31] datasets. The FDA drugs were obtained from the DrugBank database [69]. Removing redundant chemicals gave 1515 FDA-approved and 3682 KEGG-Drug chemical compounds[49]. For our purposes we used the KEGG-Drug compounds to train our model and the FDA-approved compounds to test.

For toxic compounds, we also utilized two datasets: TOXNET [68] and the Toxin Target Database (T3DB) [35]. TOXNET, maintained by the National Library of Medicine, provides toxicological datasets; specifically, we used of the Hazardous Substances Data Bank. T3DB consists of the chemical properties, molecular and cellular interactions, and medical information of several drugs and toxins. Removing redundant chemicals gave 3035 TOXNET and 1283 T3DB unique toxic compounds. For our purposes we used the TOXNET compounds to train our model and the T3DB compounds to test.

3.3 Machine Learning Setup

We utilized TensorFlow r2.3 and Keras 2.3. We made use of Google Colab GPUs for training our models. We lastly used the Python library SciPy for performing statistical analysis. We used the most up to date version of DeepSMILES while using SELFIES v0.2.4 (the version of SELFIES was updated midway through working on this project).

We used RDKit v2020.03.01 [53] to evaluate and depict molecules. RDKit is a collection of cheminformatics and machine-learning software written in C++ and Python [53]. We used GuacaMol to evaluate molecules as well. GuacaMol is an open source Python package for benchmarking model that perform *de novo* molecular design [12].

4

Experiments

4.1 Introduction

In this work, we use variational autoencoders (VAE) to create highly meaningful representations of chemical compounds. From these representations we predict several desirable properties including drug-likeness (QED), solubility ($\log P$), and synthesizability (SAS). In addition to predicting these important drug qualities, we shape the latent space of our variational autoencoders to be responsive to the transcriptomic profile of given cell lines (such that compounds with low efficacy are clustered together in one area of the latent space, while those with high efficacy are in another). Using these shaped and highly information-dense latent representations, we then predict compounds' toxicity and $\log_{10} IC_{50}$ efficacy on individual cell lines. We use transcriptomic profiles in order to represent the cell lines. The transcriptomic profiles we use are of cells *untreated* with given anti-cancer compounds from the GDSC dataset [28]. Finally, we seek to optimize compounds' efficacy against individual cancer cell lines and against particular types of cancer using Bayesian optimization (BO) and genetic algorithms (GA).

To do this, we complete five main parts: (1) the design and optimization of VAEs to capture chemical information about chemical compounds represented as SMILES [58], DeepSMILES [46], or SELFIES [34]; (2) the design of a powerful decoder to map latent variables to the space of SMILES, DeepSMILES or SELFIES; (3) the prediction of a

compound's efficacy in a disease context represented through the transcriptomic profile of cancer cells; (4) the prediction of a particular compound's toxicity from its latent representation; and finally, (5) optimization using BO and GA in a continuous latent space to discover new potential compounds with higher efficacy in particular disease contexts represented by transcriptomic profiles.

4.2 Modelling Compounds: Variational Autoencoders and Transformers

Due to the regular presence of repetitive, translation-invariant substrings (corresponding to chemical substructures like cycles and functional groups) that occur in compound representations (such as SMILES, etc.), we utilized a convolutional neural network to encode chemicals in our VAE. We mapped our SMILES/DeepSMILES/SELFIES strings to a latent space of 64 dimensions. For our VAE decoder we utilized an LSTM.

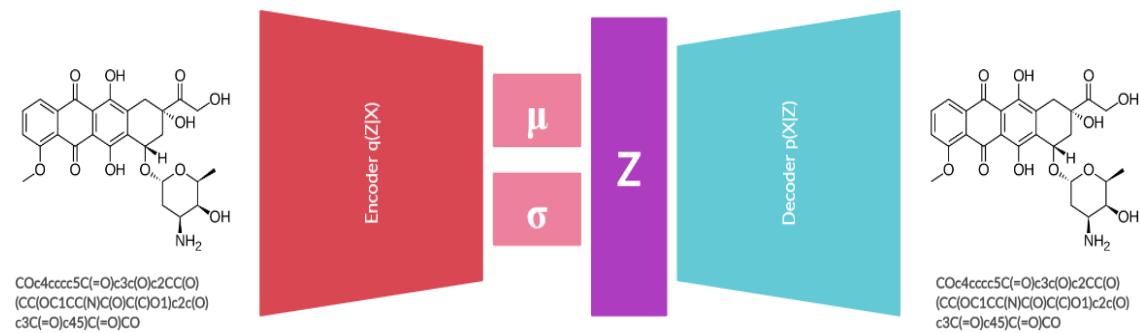


Figure 4.1: Variational autoencoder and training process. x is the discrete input vector, μ are the mean values calculated for the latent dimensions, σ are the standard deviations for the latent dimensions and z is the sampled latent representation. Encoder and Decoder are neural networks. Shows an example for the cancer drug Doxorubicin.

We trained the VAE to minimize the error in re-mapping the latent space to the original SMILES strings as seen in Fig. 4.1. The *information bottleneck* of the latent space fixed-length continuous vectors forced the VAE to learn compressed representations that captured the most salient information about the SMILES/DeepSMILES/SELFIES.

For later optimization in the latent space to succeed, most points must decode into valid SMILES chemical representations. Latent spaces often are sparse, and as a result

Hyperparameters	
Convolutional Neural Network	LSTM Decoder
Dropout: 0.2	Dropout: 0.2
Convolution Activation: Tanh	LSTM Activation: Tanh
Embedding Size : 192	LSTM Hidden Size: 256
Filter Depth Growth Rate: 1.16	Dense Layer Size: 256

Table 4.1: Hyperparameters in CNN Encoder and LSTM Decoder. For full details of the implementation see our github: <https://github.com/hanshanley/GEnerationZ>.

powerful decoders often create less meaningful latent codes; thus, the latent space often contains large "dead areas" which decode into gibberish or invalid SMILES. While VAEs encourage the development of meaningful features (see Section 2.3), posterior collapse can preclude their creation. For this reason, we adopted four different approaches to ensure that we achieved a high rate of SMILES decoding.

We firstly made use of SELFIES and DEEPSmiles. SELFIES in particular are made up of individual tokens that can be combined in seemingly arbitrary fashions to obtain syntactically valid chemical compounds. This does not mean that these compounds are chemically reasonable or readily synthesizable, however. Despite this, by exploring the use of DeepSMILES and SELFIES and we hoped to eliminate syntax as an issue.

We secondly used cyclical annealing. Cyclical annealing [20] has been reported to enable the progressive learning of more meaningful latent codes over the epochs of training.

We thirdly used the implicit representation and training of latent codes proposed by Fang *et al.* [17]. This approach utilizes sampling to ensure that the aggregated latent points match the used prior (as discussed in Section 2.3).

We lastly used a powerful transformer decoder to decode the pre-trained latent codes. This decoder works as follows: after training our VAE to discover the latent codes, all the SMILES in our datasets are mapped to their latent representations. We then trained this transformer decoder to map these latent representations back to their SMILES strings. The decoder uses the architecture designed by Radford *et al.* [50] without positional embeddings, with six decoding layers, and an embedding size of 192. In addition to these transformer layers, as in Liu *et al.* [37], we appended an LSTM to the transformer output before decoding to original discrete SMILES representations. The transformer

approach essentially utilizes the VAE to learn feature representations of compounds, and then the transformer takes responsibility for mapping compounds back to their original representation. Instead of a VAE approach, this is essentially a *feature learning* approach.

Separately, we also attempt to shape the latent space of our models to create a gradient along IC₅₀ values. We utilize Gómez-Bombarelli *et al.*'s [24] approach by jointly training our VAEs with a multi-layer perceptron neural network to predict IC₅₀ values from the transcriptomic profiles of different *untreated* cells and the latent representations of the anticancer compounds. By training for this type of property prediction (*i.e.* efficacy against a given *untreated* cell line's transcriptomic profile) the distribution of molecules in the latent space can then be organized by this property.

We now present some initial results for modelling compounds: First, we checked if our models' latent representations could be effectively and correctly mapped back into discrete chemical representations. We display decoding rates for all our trained models. Baseline models were trained with cyclical annealing and for 40 epochs. Implicit models were trained with linear annealing for 40 epochs as well. We trained all models using the ChEMBL dataset with 16,000 compounds held out for testing. In addition, in order to ascertain whether our VAE managed to effectively generate meaningful latent vectors, we checked if their latent representation were sufficiently information-dense.

4.2.1 Decoding Rates

We make a distinction between syntactically correct SMILES, DeepSMILES, and SELFIES strings and chemically reasonable SMILES, DeepSMILES, and SELFIES strings. Even though a SMILES, DeepSMILES, and SELFIES string could be syntactically correct, it may not be chemically reasonable (*i.e.* it cannot be kekulized, sanitized, or converted to a valid molecule by RDKit). As such, the "chemically reasonable decoding rate" properly gives how well models decode latent representations back to correct SMILES, DeepSMILES, or SELFIES strings. In addition to the ability of our model to decode latent representations from the ChEMBL test set, we also utilize random chemicals selected from the ZINC dataset as a further validation set.

Model	ChEMBL: Syntactic Decoding Rate	ChEMBL: Reasonable Decoding Rate	ZINC: Syntactic Decoding Rate	ZINC: Reasonable Decoding Rate
<i>Cyc VAE SMILES</i>	0.849	0.814	0.692	0.634
<i>Cyc VAE SMILES Transformer</i>	0.944	0.937	0.804	0.749
<i>Cyc VAE DeepSMILES</i>	0.958	0.775	0.893	0.705
<i>Cyc VAE DeepSMILES Transformer</i>	0.985	0.932	0.925	0.842
<i>Cyc VAE SELF-IES</i>	1.0	0.819	1.0	0.827
<i>Cyc VAE SELF-IES Transformer</i>	1.0	0.942	1.0	0.941
<i>Imp VAE-SMILES</i>	0.753	0.699	0.681	0.634
<i>Imp VAE SMILES Transformer</i>	0.831	0.814	0.737	0.707
<i>Imp VAE DeepSMILES</i>	0.976	0.801	0.829	0.684
<i>Imp VAE DeepSMILES Transformer</i>	0.950	0.828	0.913	0.819
<i>Imp VAE SELF-IES</i>	1.0	0.851	1.0	0.822
<i>Imp VAE SELF-IES Transformer</i>	1.0	0.939	1.0	0.927

Table 4.2: Decoding rate for 1000 random points in ChEMBL test set and the ZINC dataset. We make a distinction between syntactically correct SMILES and chemically reasonable SMILES. RDKit was used to check the chemical syntactic and chemically reasonableness scores. "Cyc" models denotes VAE models that were trained with cyclical annealing. "Imp" denotes models trained with implicit latent encodings. "Transformer" denotes decodings done with the transformer decoder.

As seen in Table 4.2, the decoding rates for all the models on the ChEMBL datasets are higher than on the ZINC dataset. However, by utilizing DeepSMILES and SELFIES and/or using the Transformer decoding, we see that the rate of decoding increases from the baseline VAE-SMILES model. SELFIES which offer perfect syntactic decoding further give us the best results on both the ChEMBL and ZINC datasets. All these models illustrate that our VAE model is able to effectively map unseen latent representations of chemicals from the latent space back to their corresponding discrete representations. These results also illustrate the effectiveness of the additional transformer decoder in aiding decoding from the latent space. We lastly see that we attain better results by utilizing cyclical annealing rather than implicit latent codes. Overall, the best method to generate reasonable molecules used the SELFIES representation and cyclical annealing.

4.2.2 Tanimoto Similarities

We utilize RDKit to calculate fingerprints and similarity scores. Specifically, the default RDKit Daylight fingerprint identifies all the chemical subgraphs within each molecule (where each node is an atom and each edge is a bond), hashing them to generate a "raw bit ID" [53]. The default scheme for hashing subgraphs is to hash the individual bonds based on the following:

1. the types of the two atoms in the bond;
2. the degrees of the two atoms in the path;
3. the bond type (or including aromatic bonds).

The default daylight fingerprint size is 2048 bits with a minimum path size of 1 bond and maximum path size of 7 bonds.

We first mapped 4000 random ChEMBL SMILES from our test set (*i.e.* not trained upon) to the latent space of our cyclically annealed SMILES VAE. We then reduced the dimensionality using linear PCA. As seen in Fig. 4.2, as the compounds get closer to the compound "COc3ccc(NC(=O)[C@H](C)NC(=O)C2CCN(S(=O)(=O)c1ccc(C)cc1)CC2)cc3", their Tanimoto similarity values increase. This indicates that the latent space corresponds

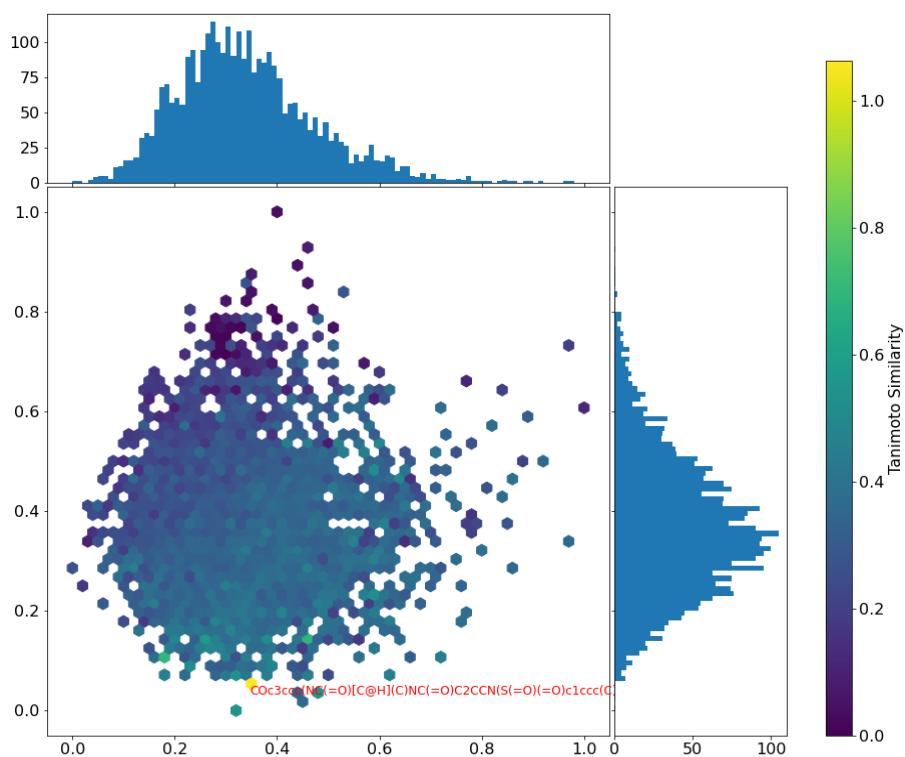


Figure 4.2: Hexbin plots and distribution histograms of the Tanimoto similarities of 4000 random chemical compounds from the ChEMBL SMILES test set against a single compound after projecting their latent representations using linear PCA. Identical pairs of molecules have a Tanimoto similarity of 1 and are coloured yellow, while less similar molecules are green, and completely dissimilar molecules are dark blue.

to the chemical composition of the test compounds, evidencing that the latent space is meaningful. We achieved similar results for the other models, see Appendix A (Figs A.1-A.5) for these graphs.

4.2.3 Compounds Near Ibuprofen in Latent Space

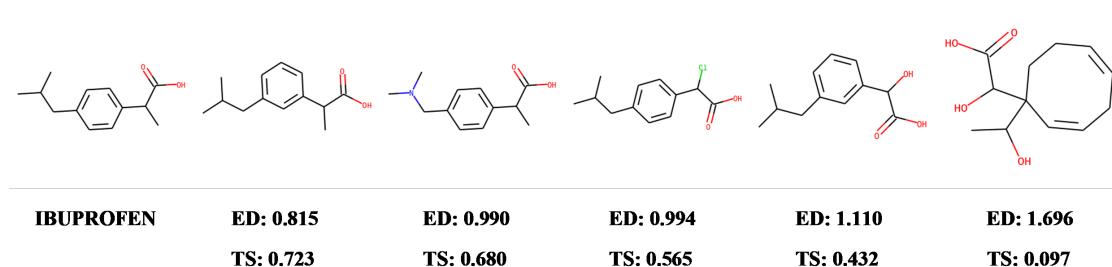


Figure 4.3: Chemical compounds in the vicinity of the Ibuprofen with their Euclidean distance (ED) in the latent space and Tanimoto Similarity (TS).

We secondly took the drug Ibuprofen and embedded it within the cyclically annealed SMILES VAE latent space. We now see from the SMILES 2-D representations in Fig 4.3 that as we move farther away from Ibuprofen’s latent space representation, the corresponding SMILES move further away in similarity further validating the usefulness of the latent space representations. We achieved similar results with the other trained models.

4.2.4 QED, logP, and SA Scores in the Latent Space

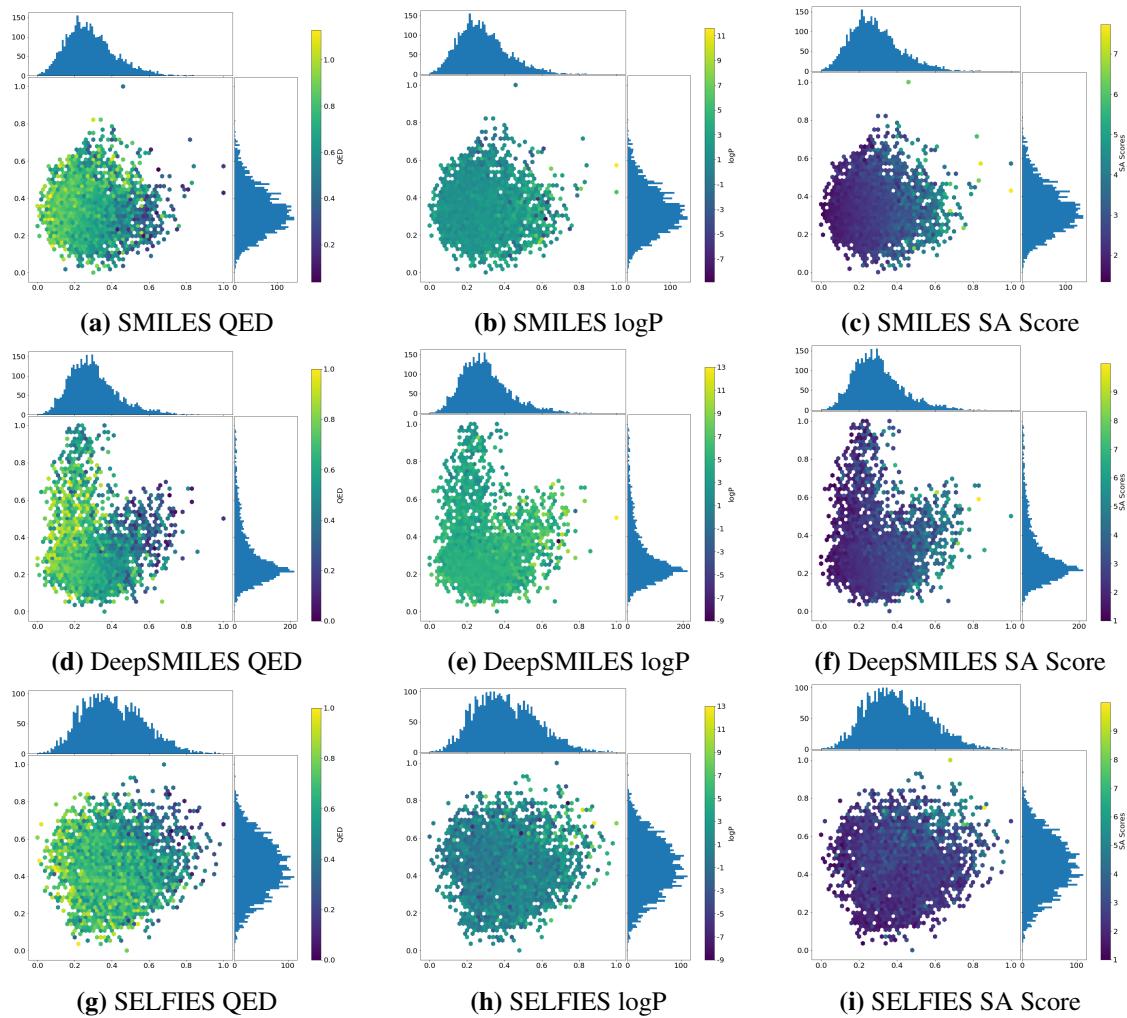


Figure 4.4: Hexbin plots of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ChEMBL test set after projecting using linear PCA the latent representations of the cyclically annealed VAEs.

In order to further understand the information-density of the latent space of our models, we plotted the drug-likeness (QED) scores of the same compounds using linear PCA on the latent space representations. We first show the results for the cyclically annealed models.

As seen in Fig. 4.4, there is a clear gradient for the QED, and SA Scores, while the logP values have a less identifiable gradient. The latent space automatically captured important qualities about the chemical compounds. This indicates that latent space is indeed meaningful. We see this behaviour across the SMILES, DeepSMILES, and SELFIES latent representations.

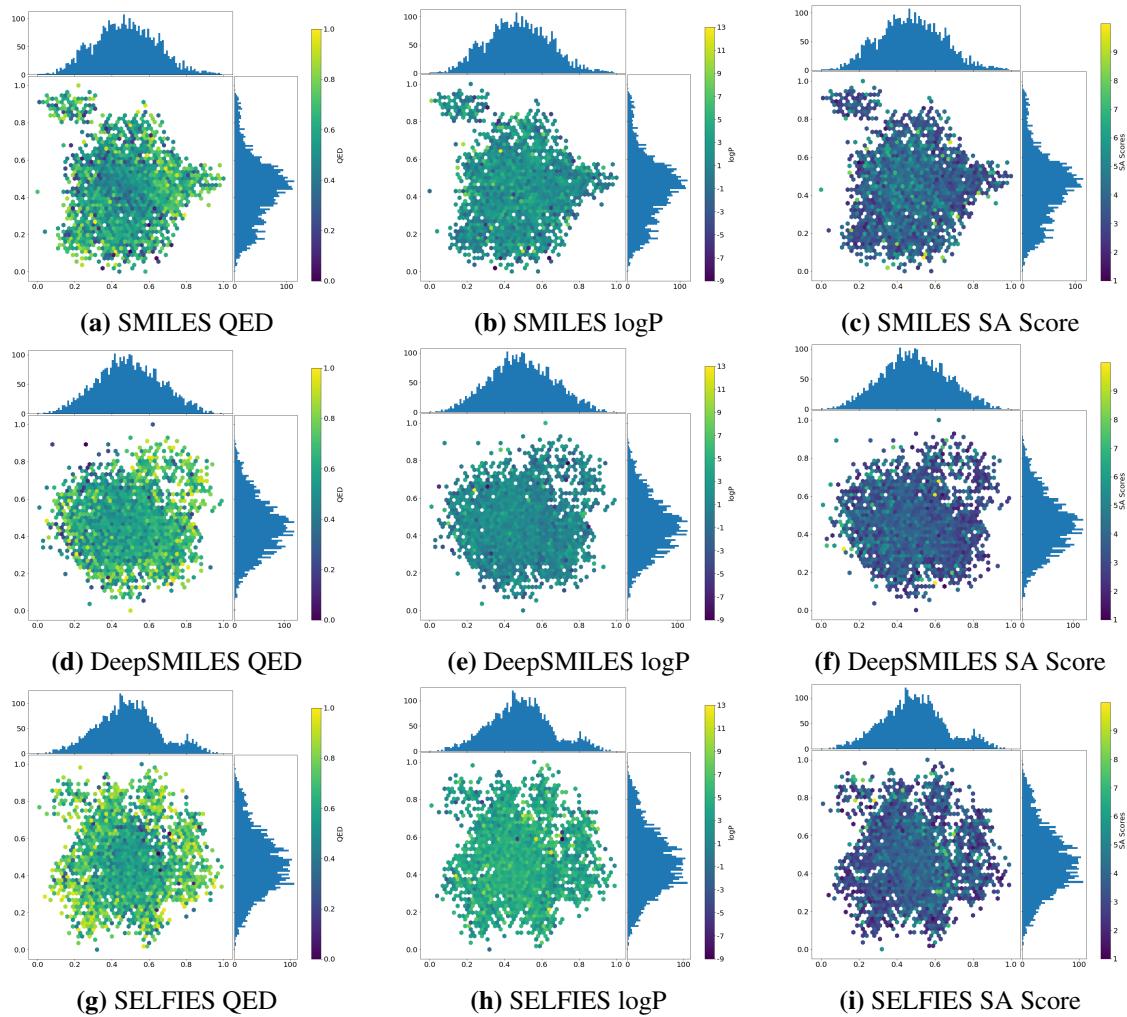


Figure 4.5: Hexbin plots of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ChEMBL test set after projecting using linear PCA the latent representations from implicit VAE models.

We now present the results for implicit code VAE models. As seen in Fig. 4.5, the models were unable to capture the same information that the cyclically annealed models did. For this reason, we did not further investigate their use in the rest of this work. We leave trying to optimize these models so that they can automatically learn the key features

of the chemical compounds to future work. In order to further investigate the information-

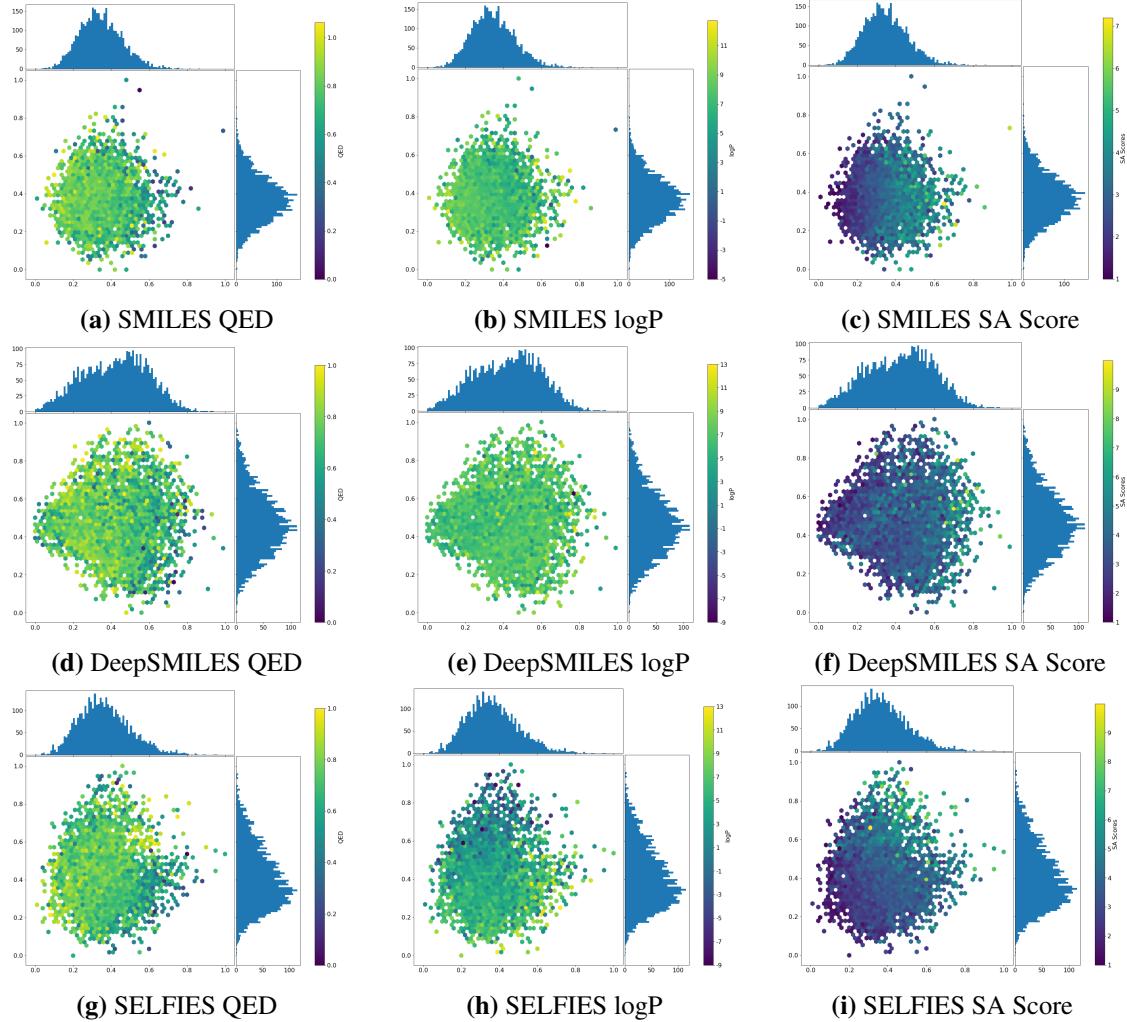


Figure 4.6: Hexplot of mean QED, logP, and SA scores of 4000 random chemicals compounds from the ZINC SMILES testing test after projecting latent representations using linear PCA.

density of the cyclically annealed latent space, we tested this approach on the ZINC database in order to ensure that our approach worked for compounds outside of ChEMBL.

Taking another 4000 random compounds from the ZINC dataset, we again see similar trends in property gradients in Fig. 4.6 for QED and SA scores for SMILES, DeepSMILES, and for SELFIES. This indicates we can elicit information-dense latent representation from all three types of input molecular representations.

4.3 Predicting $\log_{10} \text{IC}_{50}$: Attention Based Neural Networks

Although predicting and optimizing QED, logP, and SA Scores are important in their own right, we focused on optimizing IC₅₀ efficacy values for compounds against a particular cancer cell target profile. We thus developed an attention-based neural network to predict log₁₀IC₅₀ values from the latent space projection of compounds and the transcriptomes of cancer cell targets. We aim to predict log₁₀IC₅₀ values from solely their latent representations and without their corresponding complete SMILES. This allows for later optimization in the latent space according to the IC₅₀ values. We note again that we used a subset of the transcriptome, namely 2128 genes, following the approach of Manica *et al.* [41]. See Manica *et al.* [41] for details on the approach of selecting 2128 genes. We utilized IC₅₀ and SMILES drug data from the GDSC dataset for these predictions. We used 20% of the cell line pairs for testing and the rest for training and validation.

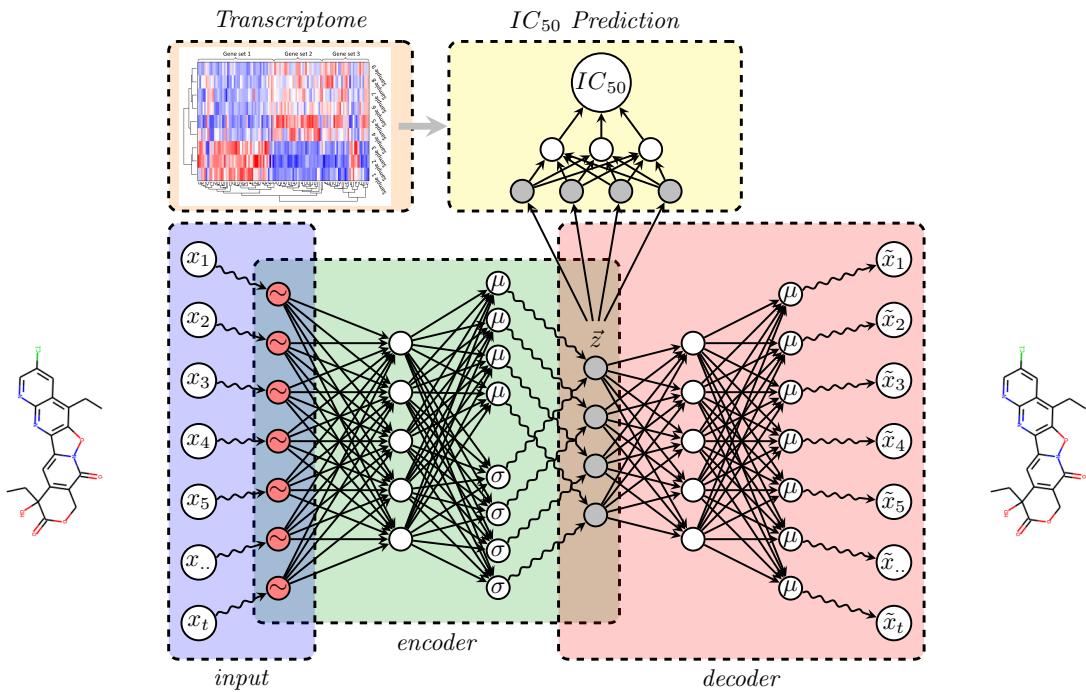


Figure 4.7: Model workflow to predict IC₅₀ .

Gene Attention and Contextual-Attention

We build our model based on the approach utilized by Manica *et al* [41]. Specifically, our modified Manica *et al.* [41] model utilized their definitions of gene attention and contextual-attention. Contextual attention allows embedding SMILES and gene expressions to interact yielding information about how individual fragments interact with genes.

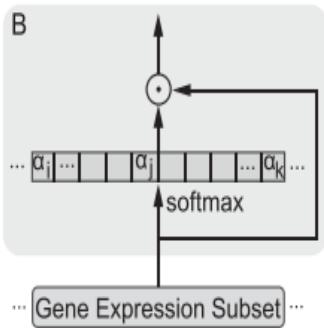


Figure 4.8: Calculation of gene embeddings. Figure from [41]. Gene subset is concatenated with itself after one version is with put through linear layer and a Softmax layer to compute an attention distribution (α_i).

We calculated gene attention using a self-attention encoder on the 2128 gene subset. This gene attention is calculated in the same manner as self-attention [41]. In contrast, the contextual attention layer takes in the SMILES embedding of a compound and genes from a cell to compute an attention distribution over the SMILES embedding within the context of the genes [41]. In our context, we do not utilize the SMILES tokens. Thus, to calculate contextual attention, we project our latent space back to an appropriately sized vector by repeating it, passing it through an LSTM and then through a dense layer (as in a standard decoder) and then perform contextual attention. Contextual attention is calculated as:

$$u_i = V^T \tanh(W_e s_i + W_g G) \text{ where } W_g \in \mathbb{R}^{A \times |G|}.$$

$$\alpha_i = \frac{\exp(u_i)}{\sum_j \exp(u_j)} \quad (4.1)$$

The matrices W_g (project genes) and W_e (project chemical embeddings) project both the genes, G , and SMILES/DeepSMILES/SELFIES embedding, s_i , to a common attention space, A . The α vector then supplies the attention vector over the partially decoded SMILES, given the gene context, G . Once the gene encodings and the contextual embeddings are computed, we utilize several dense layers before outputting the predicted $\log_{10}IC_{50}$ value. For full details see our github <https://github.com/hanshanley/GEnerationZ>.

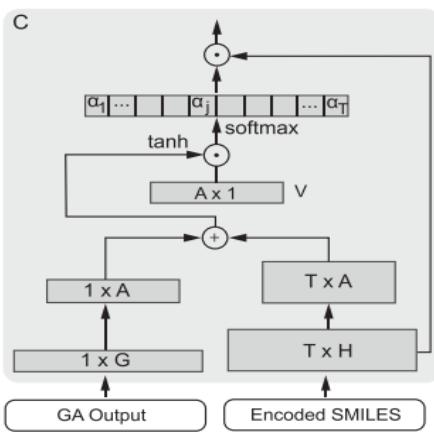


Figure 4.9: Calculation of contextual embeddings [41]. Gene attention output and encoded smiles are put through A contextual attention (CA) layer. The CA layer then outputs an attention distribution(α_i) over the SMILES encoding, in the context of transcriptomic profile.

We note here that most importantly, unlike Manica *et al.* [41], by making use of our VAE embeddings instead of SMILES strings, we are able to borrow predictive and information power from the ChEMBL dataset. We have shown that our embeddings are fairly information-dense. By utilizing them, we now show that we can achieve high performance predicting $\log_{10}\text{IC}_{50}$ values from transcriptomics data of cancer cells and SMILES. We further show that we can achieve good results without resorting to using ensembles as in Manica *et al.* [41]. We now show the $\log_{10}\text{IC}_{50}$ prediction utilizing the embeddings of each cyclically annealed VAE model:

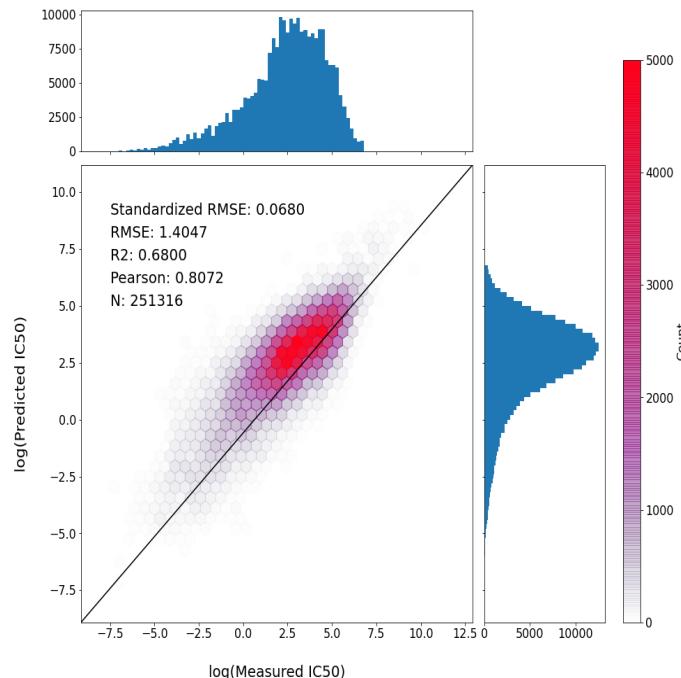


Figure 4.10: Prediction of $\log_{10}\text{IC}_{50}$ using transcriptomic data and SMILES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10}\text{IC}_{50}$ on a [0,1] scale.

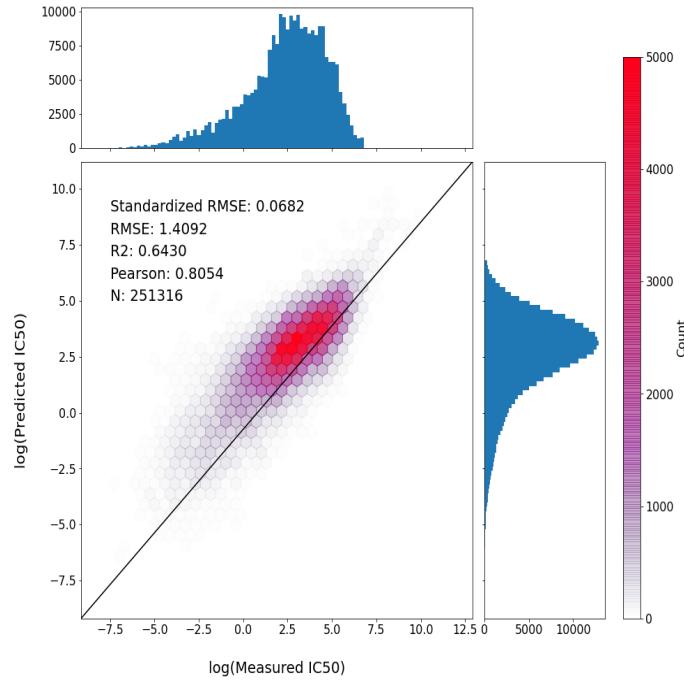


Figure 4.11: Prediction of $\log_{10}\text{IC50}$ using transcriptomic data and DeepSMILES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10}\text{IC50}$ on a $[0,1]$ scale.

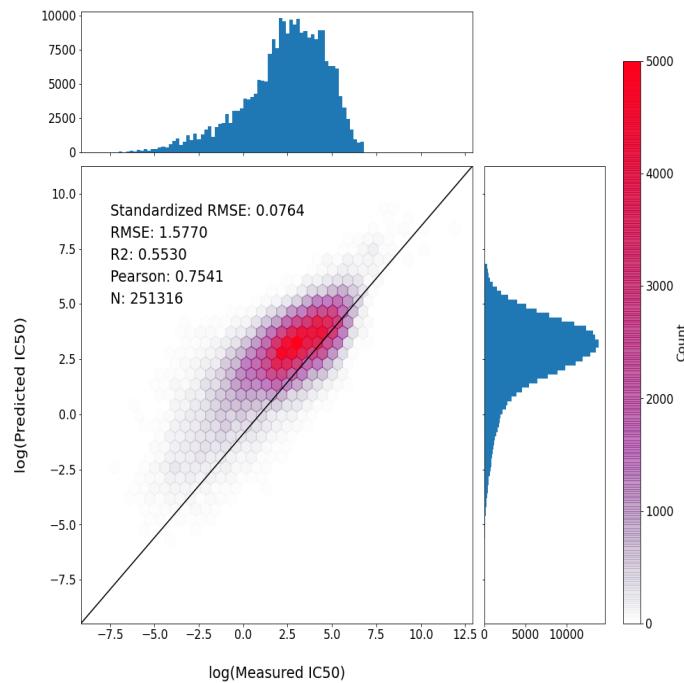


Figure 4.12: Prediction of $\log_{10}\text{IC50}$ using transcriptomic data and SELFIES latent embeddings. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10}\text{IC50}$ on a $[0,1]$ scale.

As seen in the graphs of predicted $\log_{10}\text{IC}_{50}$, we achieved fairly high Pearson coefficients utilizing our models, with the "best" model utilizing the original SMILES embeddings. Testing which one was better using one-sided Mann-Witney tests on the prediction error between the SMILES and DeepSMILES model, we found that SMILES was "significantly" better (p-value = 0.003).

4.4 Optimizing the Latent Space for Predicting $\log_{10} \text{IC}_{50}$

In order to better predict $\log_{10} \text{IC}_{50}$ values using our convolutional encoder, we shaped the latent space of our VAE to contain gradients for the $\log_{10} \text{IC}_{50}$ value of compounds conditioned on individual cell's transcriptomic profiles. To do so, we trained a multi-layer

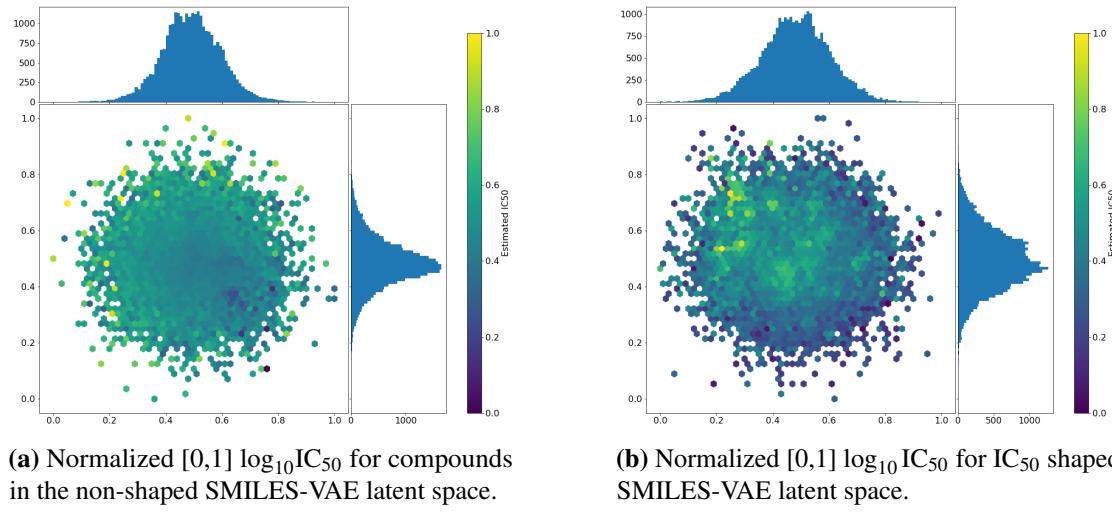


Figure 4.13: Normalized [0,1] $\log_{10} \text{IC}_{50}$ values for chemical compounds after projecting using linear PCA against the UMC-11 cell line, a cell of a carcinoid-endocrine tumour affecting the lung.

perception (MLP) jointly with our VAEs to predict $\log_{10} \text{IC}_{50}$ values. This MLP took as input the latent space representation of known cancer drugs and the transcriptomic profile of different cell lines and predicted the $\log_{10} \text{IC}_{50}$ value. Gomez-Bombarelli *et al.* [24] used this same mythology in their approach to shape the latent space according to QED, SAS, and logP values. (We also note that this approach will later also assist with optimizing the latent space for the IC_{50} value). We performed this optimization on all models but show the latent space only for SMILES models for simplicity.

Fig. 4.13 illustrates clearly that we can shape the latent space to be responsive to particular cell lines. We can organize the latent space such that efficacious compounds are in one area and ineffective compounds are in another.

Using the upgraded model with latent shaping on IC_{50} , we sought to determine whether we could better predict the normalized $\log_{10} \text{IC}_{50}$ of the GDSC dataset compared to using latent embeddings from non-shaped VAE models. We stratify our train and

test sets so that only compounds used to shape the latent space are used in only the training set of the prediction models.

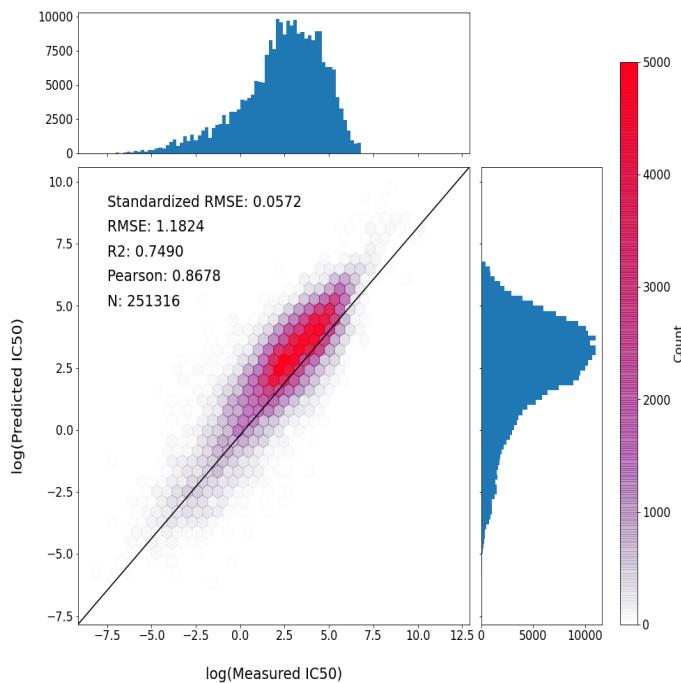


Figure 4.14: Prediction of $\log_{10} \text{IC}_{50}$ using transcriptomic data and SMILES latent embeddings with IC_{50} latent shaping. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10} \text{IC}_{50}$ on a $[0,1]$ scale.

We assess whether models using the IC_{50} shaped latent embeddings are better at predicting IC_{50} than non-shaped latent embeddings. Again, we use a one-sided Mann-Witney U-test. On comparing the SMILES embeddings, we saw $W = 2.77 \times 10^{10}$, $p\text{-value} < 2.2\text{e-}16$, for the DeepSMILES we saw $W = 2.74 \times 10^{10}$, $p\text{-value} < 2.2\text{e-}16$, and for the SELFIES we saw $W = 6.31 \times 10^{10}$, $p\text{-value} < 2.2\text{e-}16$. These tests further illustrate the usefulness of this approach in getting better error rates in predicting IC_{50} from compounds' discrete representations and transcriptomic profiles.

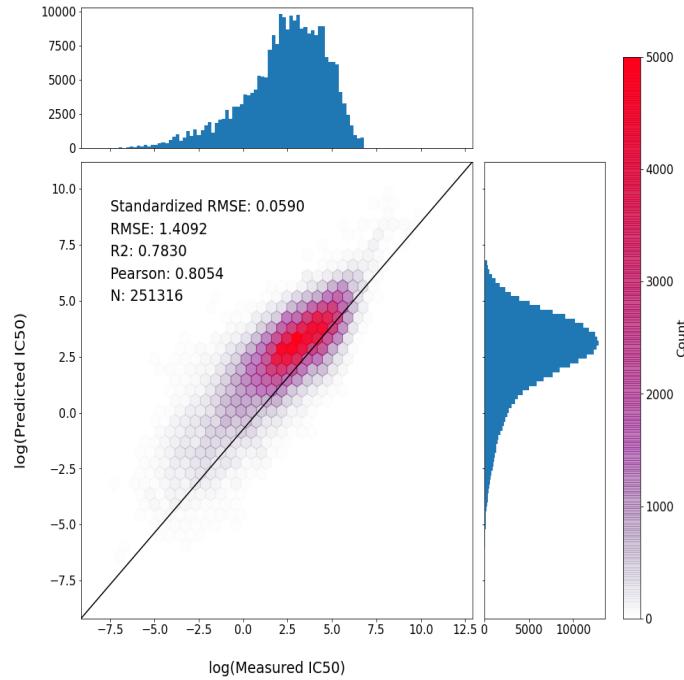


Figure 4.15: Prediction of $\log_{10} \text{IC}_{50}$ using transcriptomic data and DeepSMILES latent embeddings with IC_{50} latent shaping. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10} \text{IC}_{50}$ on a [0,1] scale.

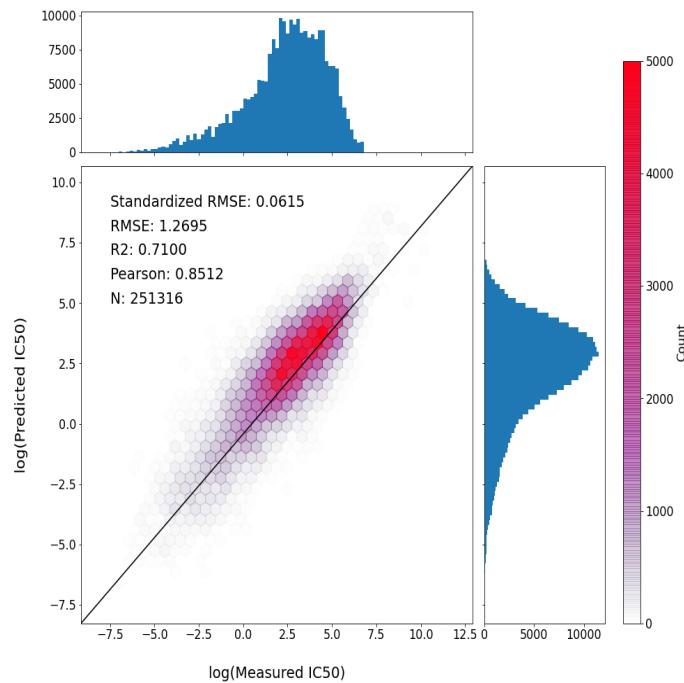


Figure 4.16: Prediction of $\log_{10} \text{IC}_{50}$ using transcriptomic data and SELFIES latent embeddings with IC_{50} latent shaping. The model was fitted in log space. RMSE was calculated after normalizing $\log_{10} \text{IC}_{50}$ on a [0,1] scale.

Model	Accuracy	TPR	FPR	F1	MCC
VAE-SMILES	0.662	0.574	0.264	0.609	0.315
VAE-SMILES <i>IC₅₀G</i>	0.668	0.595	0.270	0.622	0.328
VAE- <i>DeepSMILES</i>	0.673	0.574	0.245	0.613	0.334
VAE- <i>DeepSMILES</i> <i>IC₅₀G</i>	0.669	0.541	0.224	0.596	0.327
VAE- <i>SELFIES</i>	0.645	0.604	0.321	0.609	0.283
VAE- <i>SELFIES</i> <i>IC₅₀G</i>	0.613	0.811	0.553	0.657	0.273

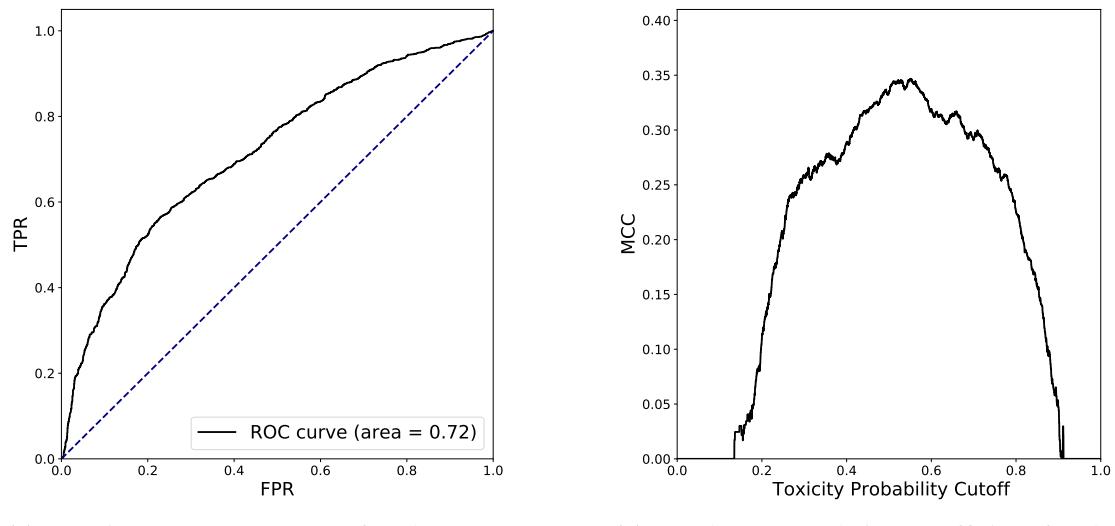
Table 4.3: Key performance metrics for the classification of chemical compounds as toxic or non-toxic. All latent embeddings were trained using cyclical annealing. The best value in each column is in bold. The IC₅₀G suffix indicates that the latent space was shaped to better predict IC₅₀ values.

4.5 Predicting Toxicity with Ensembles: Random Forests, Extra Trees, DARTs, Neural Nets, and SVMs

Lastly, we needed a means of predicting the toxicity of a compound from its latent space representation. In addition to optimizing a compound’s IC₅₀ value against particular transcriptomic profiles, we also wished to ensure that our proposed drug candidates were not highly toxic. We note the key idea is not determining exactly whether a compound is toxic or nontoxic but rather the *probability* that a given compound is toxic or not. In this way, we can screen out proposed compounds that have a high probability of being toxic. We trained models to predict toxicity using all cyclically trained models including latent models with IC₅₀ shaping.

For predicting the toxicity, we made use of a weighted ensemble of different machine learning algorithms, specifically random forests [11], extra-trees [22], DARTS [51], neural nets [66], and SVMs [61]. We used grid searches to find the optimal hyperparameters for each algorithm. We then used a weighting algorithm to find the appropriate weights for each algorithm in the full ensemble by testing on a validation set. We utilize the data from the toxicity datasets (see 3.2.4), for training, validating, and evaluating our

model. Specifically, data from DrugBank and TOXNET were used to train and validate the models while data from KEGG and T3DB were used to test the models. To predict the toxicity of compounds, we first projected them to the latent space of each particular VAE and then calculated key metrics for binary classification. We finally illustrate the MCC curve and the ROC curve of the VAE-DeepSMILES embedding model because its embeddings attained the highest accuracy of 67.3%.



(a) Receiver Operator Curve for the VAE-DeepSMILES ensemble model for predicting toxicity.

(b) Matthews Correlation Coefficient for the VAE-DeepSMILES model for predicting toxicity

Figure 4.17: Summary of the VAE-DeepSMILES embeddings for predicting toxicity.

4.6 Optimizing IC_{50} and Creating Realistic Drugs

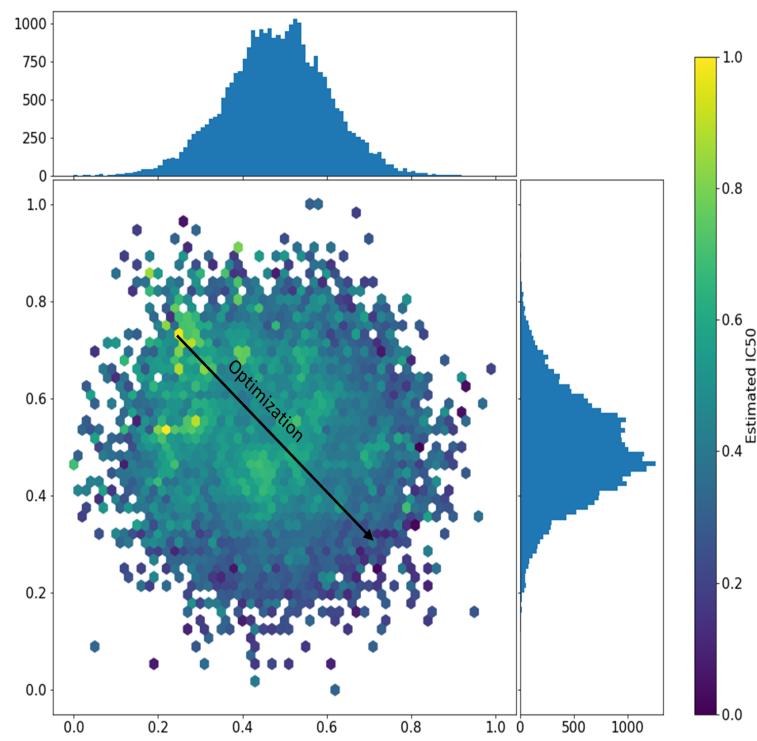


Figure 4.18: Example of optimization of normalized $[0,1] \log_{10} IC_{50}$ in the latent space.

The end goal of optimizing druglike compounds is not to find *one* particular compound with high-valued desirable qualities but rather a host of different compounds that can be later tested. Namely, in this work, we are not searching for the compound with the highest efficacy. Rather we are searching for a group of compounds in the latent space, Z , that follow a distribution, q , that have several desirable properties (high QED, low SA). This distribution q is well considered if it maximizes $E_{Z \sim q} f(Z)$ where f is a function of desirable properties. We employ two methodologies to do this: namely, Bayesian optimization and genetic algorithms. We do not encounter many of the same issues formerly addressed by Griffiths *et al.* [27]; namely, by utilizing a strong transformer decoder, cyclical annealing, and by utilizing SELFIES, we avoid the prevalence of "dead areas" in our latent space.

In Gomez-Bombarelli *et al.* [24], the primary goal was to generate new compounds that were highly synthesizable and druglike. In addition to this goal, we wish to

generate compounds with high efficacy against particular types of cancer. Given these two complementary, although somewhat different goals, we need to optimize two different objectives when generating new compounds. Thus, we first to optimize IC₅₀ of compounds against particular transcriptomic profiles in order to generate highly potent compounds. After optimizing these compounds for their IC₅₀ value, we calculate their SA scores, QED, logP, and toxicity level and then ensure that we have appropriately desirable characteristics. More specifically, we seek to ensure that the IC₅₀ metric as well as the following metric are simultaneously optimised:

$$5 \times \text{QED} - \text{SAS}, \quad (4.2)$$

This optimization metric was first proposed by Gomez-Bombarelli *et al.* [24] and has been used subsequently by others in the literature [26]. This metric indicates how likely a compound is synthesizable and drug-like.

In order to optimize compounds within the latent space, we utilize our convolutional encoder to predict the IC₅₀ values of our proposed candidate compounds. We further use the VAE-DeepSMILES embedding of the proposed compounds to compute the toxicity probability. We finally use RDKIT to predict QED, logP, and SA scores.

We wish to see whether any of the proposed candidate compounds are similar to currently approved drugs. For this reason, we also include the nearest Tanimoto similar drug from the FDA-approved/KEGG dataset and from 617 common anticancer drugs listed by Rayan *et al.* [52]. Combined, we compare each compound to 5814 unique drugs.

Lastly, we describe the distribution of the generated compounds' desirable properties compared to the compounds within the ChEMBL dataset and a list of anticancer drugs. In this way, we show how our method generates a host of unique compounds whose properties exceed those randomly and delicately chosen. We now turn to the actual optimization algorithms and our approach.

Data	N	QED	logP	SA Score	Estimated $\log_{10}IC_{50}$
ChEMBL	1.6M	0.56(0.21)	3.48(1.80)	2.89(0.76)	2.93(1.45)
ZINC	249k	0.73(0.14)	2.46 (1.43)	3.05(0.83)	3.39(1.38)
Approved Cancer Drugs	194	0.469(0.17)	3.89(1.69)	2.93(0.87)	2.58(1.91)
GA QED/SAS Opt SELFIES	100	0.91(0.04)	2.74(0.52)	2.37(0.20)	3.19(0.44)
GA QED/SAS Opt IC ₅₀ G SELFIES	100	0.92 (0.02)	2.87(0.49)	2.02(0.13)	0.94 (0.96)

Table 4.4: Summary of the number of samples generated; for datasets, this denotes the number of compounds. We show also mean and the standard deviation (in parentheses) of selected properties of the generated molecules and compare them to the properties in ChEMBL, ZINC, and approved cancer drugs which were used to initialize the optimization algorithm. The approved cancer drugs are those with known IC₅₀ values for the TE-12 cell line, a carcinoma cell take from the oesophagus.

4.6.1 Bayesian Optimization (BO)

We utilized Bayesian optimization to explore the latent space of our models and generate compounds with high efficacy (*i.e.* low log₁₀IC₅₀). We utilize a BO schema with a Gaussian Process to approximate the objective and utilize *expected improvement* for our acquisition function. We use BO to generate 1000 new points in latent space. We ran BO IC₅₀ using the cyclically annealed SELFIES embeddings as well as IC₅₀ shaped SELFIES embeddings. We refer to IC₅₀ shaped embeddings as "SELFIES IC₅₀G".

4.6.2 Genetic Algorithm (GA)

We used a genetic algorithm to explore the latent space of our models and to generate new compounds. To do so, in addition to exploring variations using Gaussian noise we combined the latent representations of known anticancer drugs with arithmetic crossover. We ran the genetic algorithm to optimize the QED/SAS metric as well as the IC₅₀ metric. After initially optimizing IC₅₀, we found that we managed to discover several potent compounds; however, simultaneously, these compounds did not have desirable QED or SA scores. For this reason, after running a genetic algorithm to optimize IC₅₀ values for 10 generations, we also ran another genetic algorithm for 5 generations to optimize

the QED/SAS metric. We ran GA IC_{50} utilizing the cyclically annealed SELFIES embeddings as well as IC_{50} shaped SELFIES embeddings.

4.6.3 Drug-likeness and Synthesizability Optimization

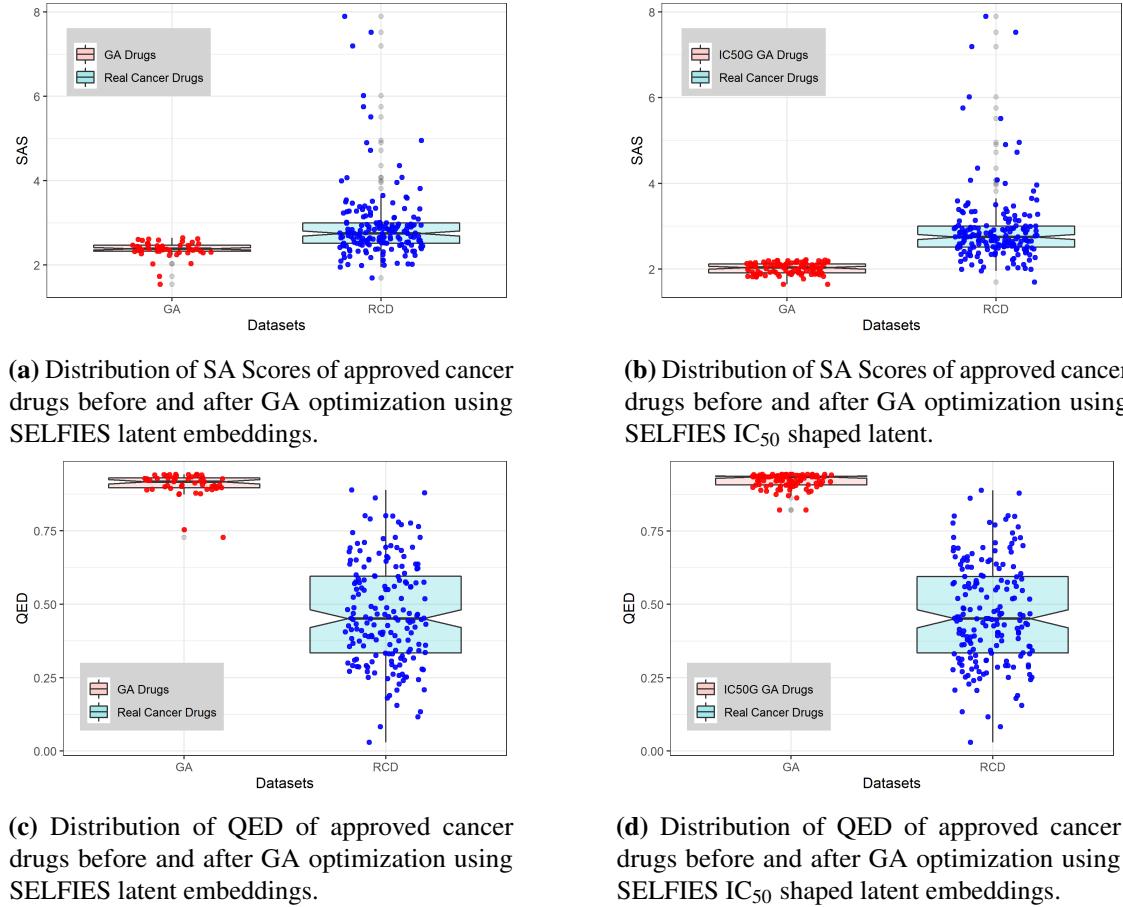


Figure 4.19: Distribution of QED and SA Scores of found cancer drugs using GA optimization.

We first present results from the optimization of the drug-likeness of generated compounds using our genetic algorithm. In Table 4.4, we summarize molecules optimized first for the SAS/QED metric using the GA metric both with SELFIES embeddings and $IC_{50}G$ SELFIES embeddings. Starting from the "Approved Cancer Drugs" positions in the latent space, we managed to generate several hundred highly druglike and synthesizable compounds. Furthermore, as seen in Table 4.4, when we used $IC_{50}G$ SELFIES embeddings, these compounds were further biased to being potent against the TE-12 cell line, a carcinoma cell found in the oesophagus. This illustrates the

usefulness of our approach (solely optimizing for QED/SAS) in generating unique and high quality druglike compounds.

This is further confirmed in plotting the QED and SAS distribution of QED/SAS optimized compounds against the distribution of approved cancer drugs. As seen in Fig. 4.19, we managed to shift the distribution of generated compounds to highly synthesizable and druglike molecules. We finish this section by displaying examples of these compounds as well as their nearest "real-life" approved counterpart in Table 4.5.

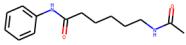
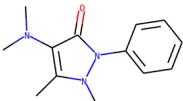
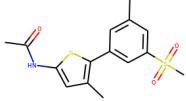
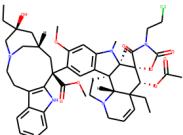
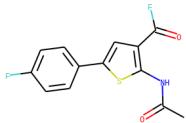
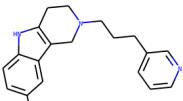
Discovered Molecule	Closest Approved Drug	Tanimoto similarity
 <p> $\text{C1=CC=C(C=C1)NC(=O)}$ CCCCNC(=O)C Est $\log_{10}\text{IC}_{50}$: 2.62 (0.45) QED: 0.727 logP: 2.32 SAS: 1.55 Tox Prob: 0.18 </p>	 <p> DB01424 $\text{CN(c1c(C)n(n(c1=O)c1ccccc1)C)C}$ Est $\log_{10}\text{IC}_{50}$: 4.14 (0.60) QED: 0.785 logP: 1.55 SAS: 2.16 </p>	0.735
 <p> $\text{CC1=C(SC(=C1)NC(=O)C)C2=CC(=CC(=C2)C)S(=O)(=O)C}$ Est $\log_{10}\text{IC}_{50}$: 2.60 (0.28) QED: 0.941 logP: 3.39 SAS: 2.51 Tox Prob: 0.11 </p>	 <p> VINZOLIDINE $\text{c12[C@@]34[C@H]}(\text{[C@]5([C@H](\text{OC(OC)=O})\text{[C]6([C@@H]3 ([N](\text{CC=C6)}\text{CC4)}) ...}}$ Est $\log_{10}\text{IC}_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.411
 <p> $\text{C1=CC(=CC=C1C2=CC(F)SC2)C(=O)F}$ Est $\log_{10}\text{IC}_{50}$: 2.97 (0.35) QED: 0.873 logP: 3.62 SAS: 2.31 Tox Prob: 0.17 </p>	 <p> D02663 $\text{Fc1ccc2c(c1)C1CN(CCCc3ccnc3)CCc1[nH]2}$ Est $\log_{10}\text{IC}_{50}$: 1.70 (1.01) QED: 0.80 logP: 3.69 SAS: 2.28 </p>	0.532

Table 4.5: Example of QED/SAS optimized molecules from the SMILES-VAE along with most similar approved drug (by Tanimoto similarity). $\log_{10}\text{IC}_{50}$ values were calculated according to sensitivity to the TE-12 cell line, a carcinoma cell in the oesophagus. We take 1000 points in the vicinity of each latent point in order to estimate the standard deviation of the estimated IC_{50} shown in parentheses. QED, logP, and SA scores were calculated using RDKit. DB=DrugBank, D=KEGG.

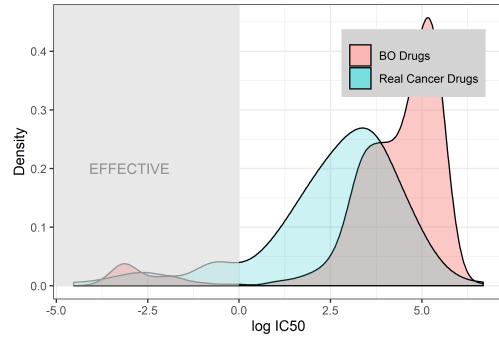
Data	N	QED	logP	SA Score	Estimated $\log_{10} IC_{50}$
ChEMBL	1.6M	0.56(0.209)	3.48(1.80)	2.89 (0.76)	2.93(1.45)
ZINC	249k	0.73(0.14)	2.46 (1.43)	3.05(0.83)	3.39(1.38)
Approved Cancer Drugs	194	0.469(0.17)	3.89(1.69)	2.93(0.87)	2.58(1.91)
BO SELFIES	1000	0.38(0.17)	4.78(1.65)	3.21(1.33)	2.45(0.85)
BO $IC_{50}G$ SELFIES	1000	0.47(0.16))	0.95(1.49)	4.92(0.81)	0.89(0.38)
GA SELFIES $IC_{50} Opt$	100	0.17(0.06)	5.02(1.11)	5.66(0.38)	-4.59(0.08)
GA SELFIES $IC_{50}/QED/SAS$ Opt	100	0.75(0.05)	3.01(1.07)	3.99(.27)	0.32(4.13)
GA SELFIES $IC_{50} Opt IC_{50}G$	100	0.39(0.06)	3.75(0.77)	3.98(0.42)	-4.67 (0.03)
GA SELFIES $IC_{50}/QED/SAS$ Opt $IC_{50}G$	100	0.77 (0.2)	3.66(0.59)	3.13(.13)	-0.78(2.67)

Table 4.6: Summary of the number of samples generated for comparison; for data, this value simply reflects the size of the data set. We show mean and the standard deviation (in parentheses) of selected properties of the generated molecules and compare them to the properties in ChEMBL, ZINC, and approved cancer drugs which were used to initialize the optimization algorithm. The approved cancer drugs are those with known IC_{50} values for the TE-12 cell line, a carcinoma cell taken from the oesophagus.

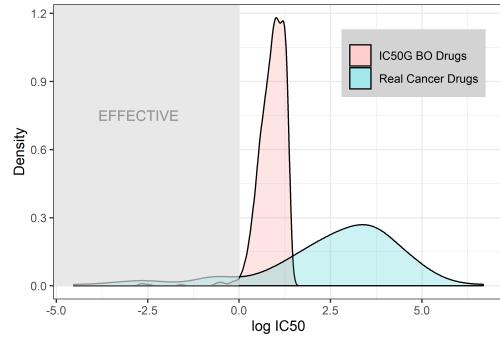
4.6.4 IC_{50} Optimization Against the TE-12 Cell Line

For IC_{50} optimization we chose to generate compounds that were potent against the TE-12 cell line, a carcinoma cell in the oesophagus (this was merely the first cell line in our dataset).

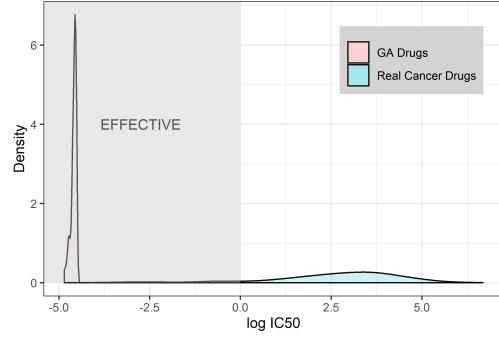
We found that BO generated a wide diversity of points when optimizing for IC_{50} as seen in Table 4.6. Specifically, by utilizing $IC_{50}G$ SELFIES we automatically generate highly potent compounds. While in ChEMBL and ZINC, estimated $\log_{10} IC_{50}$ is near 3.0, within the $IC_{50}G$ SELFIES BO data, the average estimated $\log_{10} IC_{50}$ is only 0.89. However, when we do not use $IC_{50}G$ SELFIES, we do not obtain highly potent compound, with an average $\log_{10} IC_{50}$ of 2.45. This illustrates the necessity of shaping the latent space, if BO is to be used.



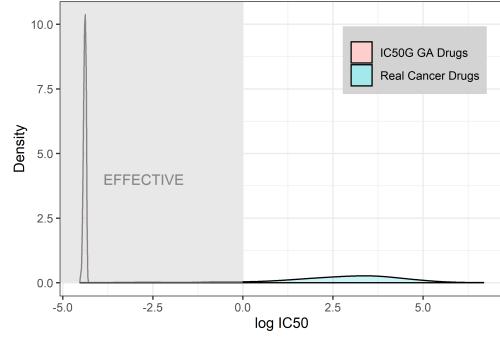
(a) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after IC_{50} BO using SELFIES latent embeddings.



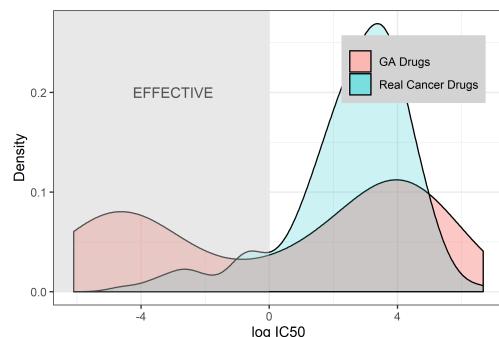
(b) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after IC_{50} BO using SELFIES IC_{50} shaped latent embeddings.



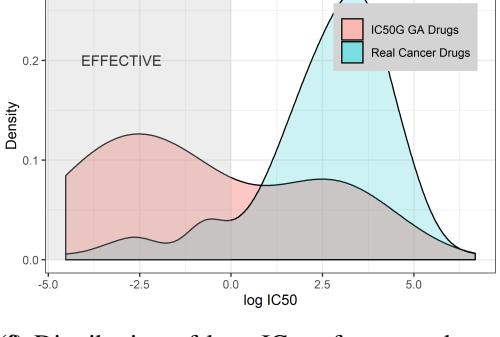
(c) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after GA IC_{50} optimization using SELFIES latent embeddings.



(d) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after GA IC_{50} optimization using SELFIES IC_{50} shaped latent embeddings.



(e) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after GA IC_{50} /SAS/QED optimization using SELFIES latent embeddings.



(f) Distribution of $\log_{10}\text{IC}_{50}$ of approved cancer drugs before and after GA IC_{50} /SAS/QED optimization using SELFIES IC_{50} shaped latent embeddings.

Figure 4.20: Distribution of $\log_{10}\text{IC}_{50}$ values in discovered cancer drugs using GA optimization compared to clinically approved cancer drugs.

We finally see that by utilizing our GA algorithm that we are able to generate hundreds of highly efficacious compounds. Namely, the average $\log_{10} \text{IC}_{50}$ using both the IC_{50}G SELFIES latent embeddings and the regular SELFIES is near -4.6. However, as seen in both cases where we just optimized IC_{50} values, our SAS and QED values are relatively poor. We followed IC_{50} optimization by five generations of SAS/QED optimization and found that we can increase the diversity of points and generate potent yet synthesizable compounds.

Furthermore as illustrated in Fig. 4.20, the distribution of predicted $\log_{10} \text{IC}_{50}$ is pushed further to being potent in every model compared to real cancer drugs (except in the case of using BO SELFIES). This further supports utilizing this approach to create highly diverse and efficacious compounds that target a specify cell line.

We finish this section by showing a sampling of generated compounds utilizing our approach. We show compounds that were generated only utilizing IC_{50} optimization and those generated using both IC_{50} and SAS/QED optimization.

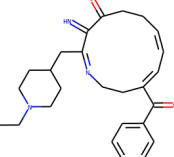
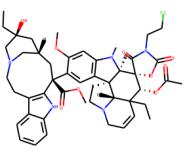
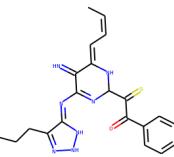
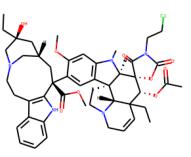
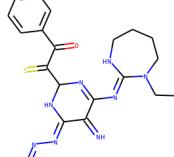
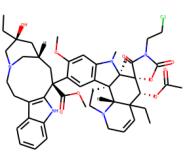
Discovered Molecule	Closest Approved Drug	Tanimoto similarity
 <p> <chem>CCN1CCCCNC1=N/C(=NC(NS3=N/N=C=S)C(=O)C2=CC=CC=C2)C3=N</chem> Est $\log_{10} IC_{50}$: -6.44 (0.53) QED: 0.13 logP: 2.20 SAS: 5.21 Tox Prob: 0.10 </p>	 <p> VINZOLIDINE <chem>c12[C@@]34[C@H]([C@]5([C@H](OC(=O)[C]6([C@H]3([N](CC=C6)CC4))CC))</chem> Est $\log_{10} IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.561
 <p> <chem>CC1=C(SC(=C1)NC(=O)C)C2=CC(=CC(=C2)CS(=O)(=O)CCCCC1=NNNC1=N/C(=NC(NC3=C/C=C C)C(=S)C(=O)C2=CC=CC=C2)C3=N</chem> Est $\log_{10} IC_{50}$: -6.50 (0.70) QED: 0.423 logP: 2.65 SAS: 4.63 Tox Prob: 0.14 </p>	 <p> VINZOLIDINE <chem>c12[C@@]34[C@H]([C@]5([C@H](OC(=O)[C]6([C@H]3([N](CC=C6)CC4))CC))</chem> Est $\log_{10} IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.770
 <p> <chem>CCN1CCCCNC1=N/C(=NC(NS3=N/N=C=S)C(=O)C2=CC=CC=C2)C3=N</chem> Est $\log_{10} IC_{50}$: -6.39 (0.09) QED: 0.134 logP: 1.92 SAS: 4.74 Tox Prob: 0.10 </p>	 <p> VINZOLIDINE <chem>c12[C@@]34[C@H]([C@]5([C@H](OC(=O)[C]6([C@H]3([N](CC=C6)CC4))CC))</chem> Est $\log_{10} IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.532

Table 4.7: Example of IC_{50} optimized molecules from the SMILES-VAE along with most similar approved drug (by Tanimoto similarity). $\log_{10} IC_{50}$ were calculated according to sensitivity to the TE-12 cell line, a carcinoma cell in the oesophagus. We take 1000 points in the vicinity of each latent point in order to estimate the standard deviation of the estimated $\log_{10} IC_{50}$ shown in parentheses. QED, logP, and SA scores were calculated using RDKit.

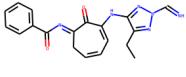
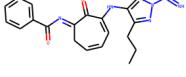
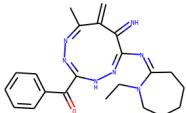
Discovered Molecule	Closest Approved Drug	Tanimoto similarity
 <p> <chem>CC/C(=N)N1C(=N)C3=C/C=C/C/C(=N)C(=O)C2=CC=CC=C2C3=O</chem> Est $\log_{10} IC_{50}$: -5.81 (0.76) QED: 0.626 logP: 2.40 SAS: 3.67 Tox Prob: 0.073 </p>	 <p> VINZOLIDINE <chem>c12[C@H]3[C@H](C1)[C@H]([C@H]1C(C(=O)OC)C)C[C@H]3[C@H]([C@H]3C(C(=O)OC)C)C</chem> Est $\log_{10} IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.504
 <p> <chem>CCCC(=N)N1C(=N)C3=C/C=C/C/C(=N)C(=O)C2=CC=CC=C2C3=O</chem> Est $\log_{10} IC_{50}$: -4.61 (2.38) QED: 0.745 logP: 3.74 SAS: 3.64 Tox Prob: 0.071 </p>	 <p> BLEOMYCIN A2 <chem>O[C@H]([C@H](NC(c1nc([C@H](NC[C@H](C(N)=O)N)CC(N)=O)nc(c1C)N)=O)C(N[C@H]</chem> Est $\log_{10} IC_{50}$: 4.61 (1.14) QED: 0.01 logP:-8.71 SAS: 7.39 </p>	0.499
 <p> <chem>CCN1CCCCC1=N/C(=N)N/N=C(C)C3=C/C(=O)C2=CC=CC=C2C3=N</chem> Est $\log_{10} IC_{50}$: -5.33 (0.62) QED: 0.753 logP: 3.43 SAS: 4.10 Tox Prob: 0.10 </p>	 <p> VINZOLIDINE <chem>c12[C@H]3[C@H](C1)[C@H]([C@H]1C(C(=O)OC)C)C[C@H]3[C@H]([C@H]3C(C(=O)OC)C)C</chem> Est $\log_{10} IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.532

Table 4.8: Example of IC_{50} /QED/SAS optimized molecules from the SMILES-VAE along with most similar approved drug (by Tanimoto similarity). $\log_{10} IC_{50}$ were calculated according to sensitivity to the TE-12 cell line, a carcinoma cell in the oesophagus. We take 1000 points in the vicinity of each latent point in order to estimate the standard deviation of the estimated $\log_{10} IC_{50}$ shown in parentheses. QED, logP, and SA scores were calculated using RDKit.

Data	N	QED	logP	SA Score	Estimated $\log_{10} IC_{50}$
Approved Cancer Drugs	203	0.47(0.17)	3.88(1.70)	2.92(0.84)	1.93(2.21)
GA SELFIES IC_{50} Opt $IC_{50}G$	100	0.45(0.12)	3.36(0.81)	3.92(0.38)	-4.84(0.03)

Table 4.9: Summary of the number of samples generated. We show mean and the standard deviation (in parentheses) of selected properties of the generated molecules and compares them to the mean and standard deviation of properties for generated compounds as well as the approved cancer drugs which were used to initialize the optimization algorithm.

Lastly, we show compounds that were optimized against a group of cell lines rather than just one. We show these compounds that were optimized with the GA algorithm against sarcoma cancer cells. This illustrates our methods’ ability to generate compounds that can be effective against an entire cancer types. We ran only our GA algorithm to generate highly potent compounds on this type of cancer. As seen in Table 4.10 and

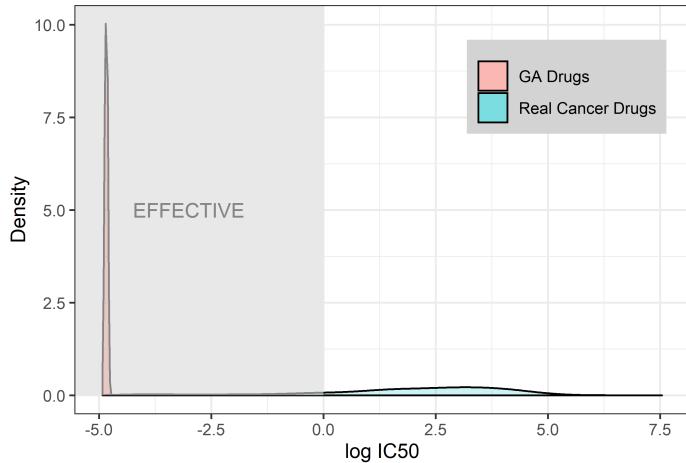


Figure 4.21: Distribution of $\log_{10} IC_{50}$ of clinically approved cancer drugs and *de novo* compounds proposed by GA optimization using SELFIES IC_{50} shaped latent embeddings.

Fig. 4.21, we can generate over 100 unique drugs with high predicted efficacy for a variety of sarcoma cell lines. This attempt at discovering new drugs for a cancer type in particular illustrates the flexibility of our approach. With more data on particular types of cancer, this approach could be further tailored and generate compounds that can better address the needs of a given cancer.

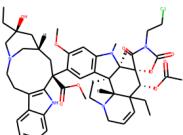
Discovered Molecule	Closest Approved Drug	Tanimoto Similarity
 <p> <chem>CCC1=C2C=C(C=NC2=NC3=C1ON4C3=CC5=C(C4=O)COC(=O)C5(CC)OCl</chem> Est $\log_{10}IC_{50}$: -4.80 (0.01) QED: 0.51 logP: 2.35 SAS: 3.40 Tox Prob: 0.11 </p>	 <p> Vinzolidine <chem>c12[C@H]34[C@H]([C@H]5([C@H](OC(C)=O)[C@H]6([C@@H]3[N]CC=C6)CC4))</chem> Est $\log_{10}IC_{50}$: 1.99 (0.97) QED: 0.13 logP: 5.04 SAS: 7.65 </p>	0.785

Table 4.10: Example of sarcoma IC₅₀ optimized molecules from the SMILES-VAE along with most similar approved drug (by Tanimoto similarity). We take 1000 points in the vicinity of each latent point in order to estimate the standard deviation of the estimated IC₅₀ shown in parentheses. QED, logP, and SA scores were calculated using RDKit.

5

Discussion

5.1 Modelling Compounds

We found that modelling compounds using VAEs was highly effective. We managed to elicit the logP, QED, and SA scores implicitly in our latent space from compounds by utilizing cyclical annealing. Unlike Gómez-Bombarelli *et al.* [24], we did not have to shape the latent space in order for our models to be responsive to these chemical properties. This illustrates that our convolutional VAEs were able to effectively learn important chemical information from our datasets without it having to be explicitly programmed. Similarly, we found that similar compounds were also grouped together within our latent space as measured by Tanimoto scores, further showing the information-density of VAEs trained using cyclical annealing.

We found that the latent spaces could also be trained to be sensitive to particular transcriptomic profiles. As shown in Fig 4.13, we can engineer a gradient for potency expressed as IC₅₀ values against a cancerous cell. This illustrates the utility of our approach in creating efficient mechanisms for later optimization of IC₅₀ values.

5.2 Predicting Toxicity

We can approximate toxicity of various approved drugs by making use of their latent embeddings. Using the coordinates of these compounds in our latent space as input

features, we show that an ensemble of machine learning algorithms can effectively predict toxicity levels. This ensemble allowed us to get a smooth prediction curve of whether a compound is toxic or not. While we were unable to obtain high accuracy for our model, we still were able to approximate the probability that a given compound is toxic using this methodology. A key area of future work would be to collect a larger dataset of approved drugs and toxic compounds that would allow us to gain a better approximation of toxicity. We note, however, that in our work, we used approved anticancer drugs as launching points within the latent space to find new compounds. As a result, we found that many of our proposed compounds had a relatively low probability of being toxic. Because we considered any drug that was approved to be "non-toxic", drugs that are similar to these already approved drugs are also likely to be considered non-toxic. Further investigation of approved toxicity prediction is need with additional criteria for considering a drug toxic.

Nevertheless, we note in conclusion that by making use of toxicity prediction, we can effectively screen out compounds that have toxic-like characteristics before further investigating them, speeding up the drug discovery process.

5.3 Predicting $\log_{10} \text{IC}_{50}$

We found in this work that by shaping the latent space to be sensitive to the RNA expressions of given cell lines that we could better predict the $\log_{10} \text{IC}_{50}$ of various compounds against these same cell lines. This makes sense given the gradient that can be engineered in the latent space towards IC_{50} efficacy as seen in Fig 4.13. Furthermore, by making use of these same embeddings, we borrowed powerful predictive power from the wide range of other chemical substances that exist within the ChEMBL training dataset. As shown in Fig 4.4, characteristics of the compounds can be automatically learned from training the VAE. As a result, information that might pertain to a compound's efficacy such as its drug-likeness (QED), which is automatically embedded within the latent space, helped in the prediction. This is a form of *transfer learning* and is a powerful direction forward in better predicting $\log_{10} \text{IC}_{50}$ values. This could also further help in designing new compounds with more desirable properties. By better combining information about

other compounds and creating ever more dense representations of compounds in the latent space, we hypothesize that more accurate predictions of $\log_{10} \text{IC}_{50}$ can be achieved.

We note here, however, that these advancements will be limited by underlying epistemic error in IC_{50} values. Because IC_{50} are manually measured in the laboratory there is inherent error in predicting these values, which limits our ability to make accurate predictions. A complete evaluation of this error is beyond the scope of this work but is an area of future work.

Manica *et al.* [41] utilized an ensemble-based approach to better predict the $\log_{10} \text{IC}_{50}$. By taking stopping their model at different times during training, and then averaging the predicted IC_{50} scores, they were able to get a higher R^2 correlation than our approach. This is an area of future work for our work, as we expect that utilizing this approach could lead to a higher accuracy and a lower RMSE.

We also wish to further make use of Daylight fingerprints in our model. As found by Manica *et al.* [41], counts of atoms and bounds are predictive in estimating a drug’s efficacy. Since Daylight fingerprints inherently contain this information, we wish to see if including that information directly in our model will improve results.

Lastly, prediction of IC_{50} values inherently falls short due to the lack of volume of high-quality data. One important next step is to acquire more reliable IC_{50} data so that we can train even more powerful models.

5.4 Optimizing Compounds

We found that both Bayesian optimization with *expected improvement* as the acquisition function, and genetic algorithms can both be effective means of optimizing compounds for IC_{50} . In particular, because the latent space is amenable to adding Gaussian noise and performing arithmetic crossover in order to move in the chemical space that genetic algorithms were particularly effective. Unlike in previous works [24], we do not need to rely on domain knowledge in our approach to perform genetic algorithms.

We see that our approach can generate a host of unique compounds that are tailored for specific cell lines and types of cancer. We further see the necessity of utilizing IC_{50} -shaping in order to generate diverse, but highly efficacious compounds. Again, by having

more data on particular types of cancer, we believe that our approach can be used to generate a wide range of new compounds that target that particular cancer. Lastly, we show the utility of our approach in generating compounds that target particular cells. Our approach highlights a possible future path in personalized medicine.

5.5 Google Colab Issues

We lastly note that in training our models, we ran into the issue of Google Colab timeouts. Although we paid for a premium subscriptions, the maximum training time of all our models was limited to 24 hours. For some of our models, this meant fewer epochs of training. We wish in the future to train our models for longer using a system with more RAM and a dedicated service.

Appendices

A

Additional Graphs

A.1 Tanimoto Similarities

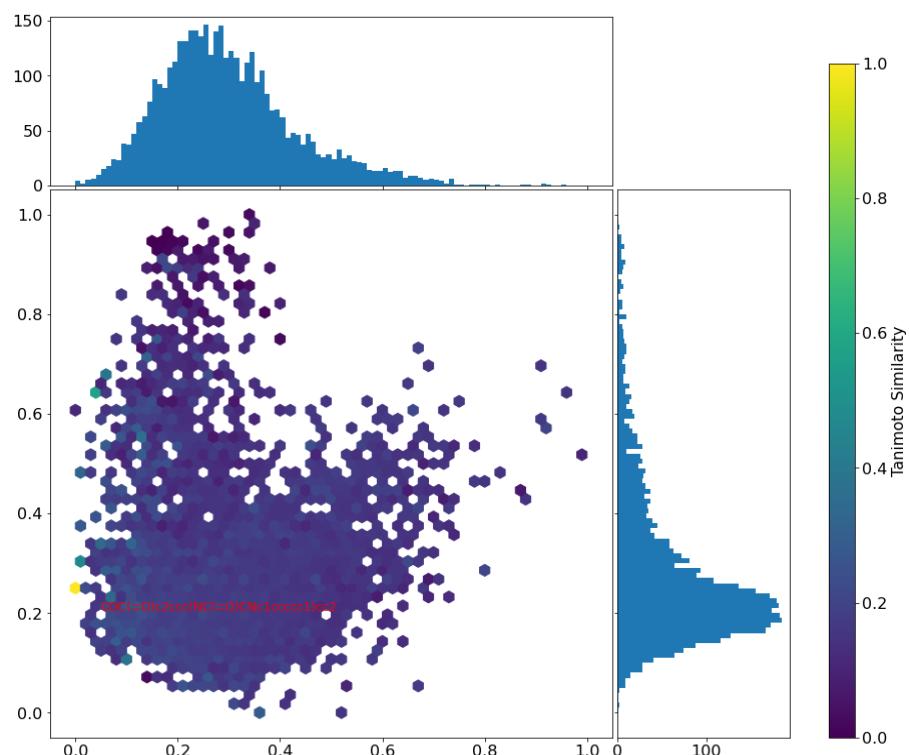


Figure A.1: Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA DeepSMILES latent representations.

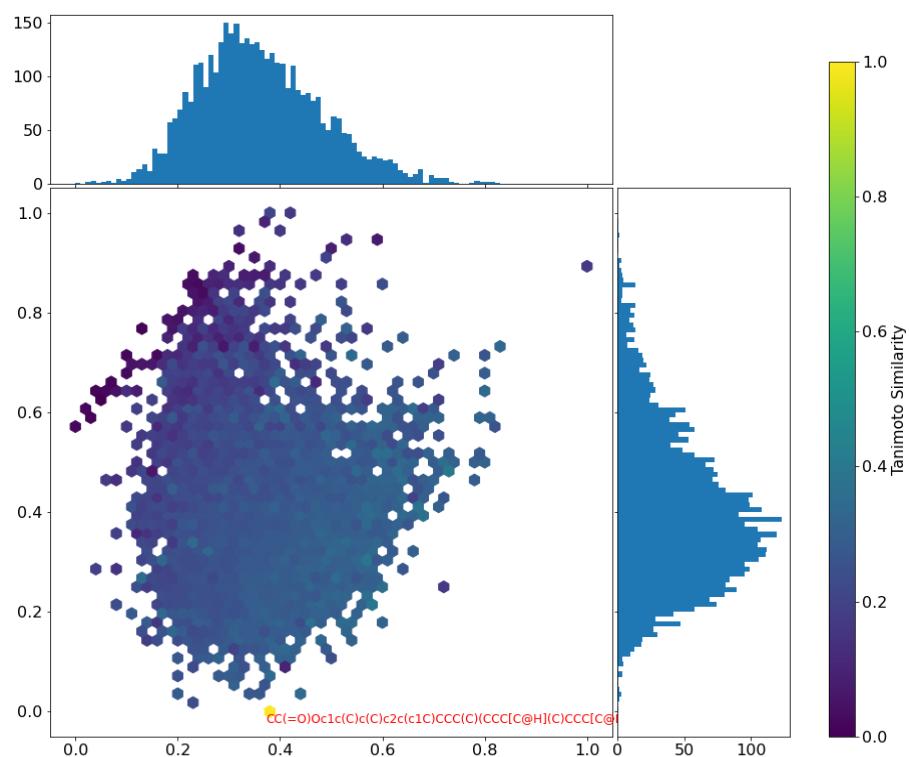


Figure A.2: Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA SELFIES latent representations.

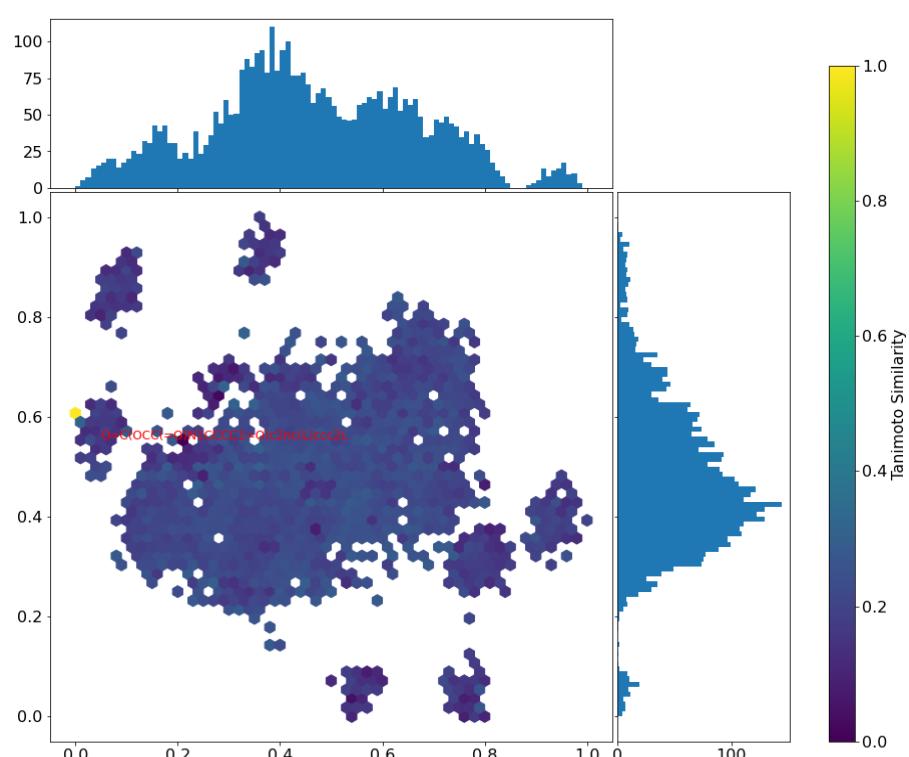


Figure A.3: Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit SMILES latent representations.

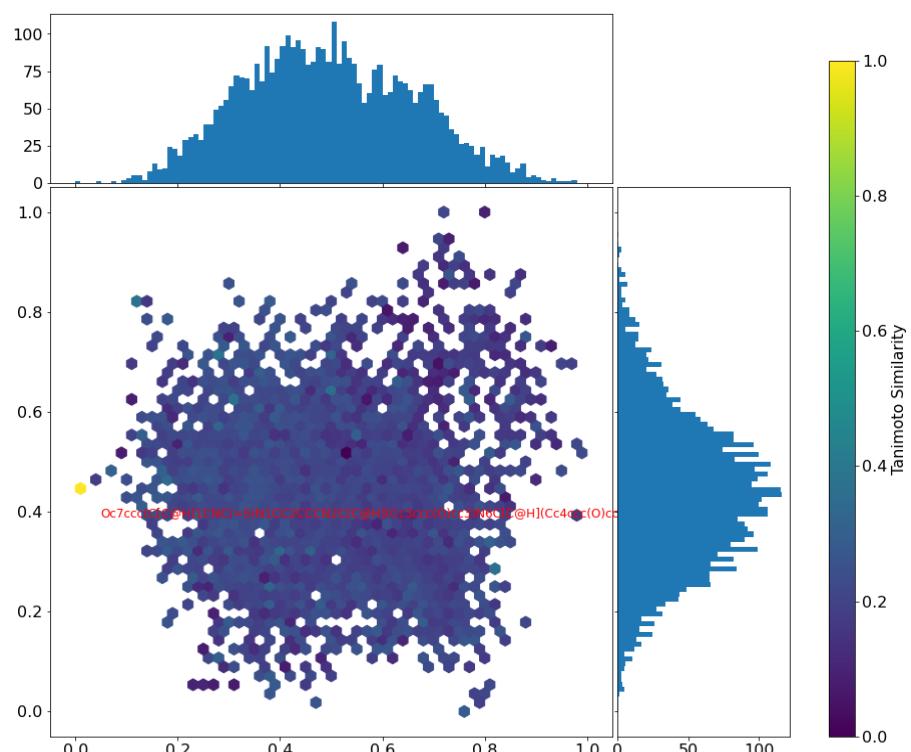


Figure A.4: Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit DeepSMILES latent representations.

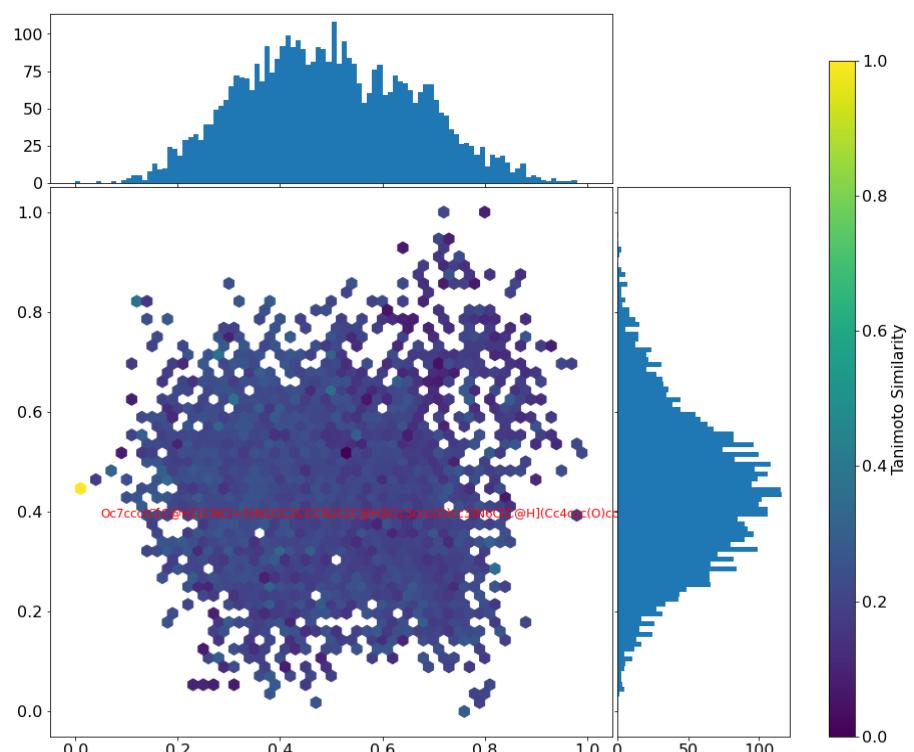


Figure A.5: Tanimoto similarities of 4000 random chemicals compounds from the ChEMBL test set against a single compound after projecting using linear PCA Implicit SELFIES latent representations.

B

Selected Code

Most of the code can be found on <https://github.com/hanshanley/Private-De-Novo-Drug-Creation>. We provide a selection of code for reference here.

Train_VAE

September 6, 2020

```
[ ]: # Initialize drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

[ ]: # Move to Google Drive
%cd drive
%cd 'My Drive'
%cd 'MSc Stats Dissertation'

[ ]: import sys
import os
sys.path.append('/usr/local/lib/python3.7/site-packages/')

[ ]: import numpy as np
import tensorflow as tf
import tensorflow.keras.backend as K
import tensorflow.keras as keras
import pandas as pd
import math
import tensorflow.keras.layers as layers
import deepsmiles
import rdkit
import selfies
import time
import numpy as np
import matplotlib.pyplot as plt

[ ]: BATCH_NORM = True
CONV_ACTIVATION = 'tanh'
CONV_DEPTH = 4
CONV_DIM_DEPTH = 32
CONV_DIM_WIDTH = 16
CONV_D_GF = 1.15875438383
CONV_W_GF = 1.1758149644
HIDDEN_DIM = 256
```

```
[ ]: def softmax_logits_loss_with_pad(labels,logits):
    weights = tf.cast(tf.not_equal(labels, 0), tf.float32)
    nonpad_seq = tf.math.count_nonzero(weights, dtype=tf.dtypes.float32, )
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=logits)
    loss = loss *weights
    return tf.reduce_sum(loss)
```

```
[ ]: ## Necessary CONSTANTS
BATCH_SIZE = 256
VOCAB_SIZE = len(vocab_index)
EPOCHS = 40
LEARNING_RATE = 0.000312087049936
DROP_OUT= 0.2
EMBEDDING_DIM = 192 ## Embedding dim of the characters
MAX_LEN = 101 ## Maximum size of a SMILE (100 + BOS, EOS)
PAD_LEN = 250
MAX_LEN = PAD_LEN - 1
DROPOUT = 0.2
TRAIN = True
```

```
[ ]: ## Import Necessary Data for training
train_smiles_X = np.load('./vocab/train_deep_smiles_X.npy',allow_pickle=True)
vocab =np.load('./vocab/deep_vocab.npy',allow_pickle=True)
vocab_index = np.load('./vocab/deep_vocab_index.npy',allow_pickle=True)
vocab = dict(vocab.ravel()[0])
vocab_index = dict(vocab_index.ravel()[0])
```

```
[ ]: index = np.where(train_smiles_X == 1)
t = np.split(train_smiles_X,index[0].tolist())
t= t[1:]
t = tf.keras.preprocessing.sequence.
    pad_sequences(t,maxlen=PAD_LEN,padding='post')
```

```
[ ]: NUM_BATCHES = math.floor(len(t)/BATCH_SIZE )
```

```
[ ]: NUM_TRAIN_BATCH = math.floor(NUM_BATCHES*0.99)
NUM_TEST_BATCH = math.floor(NUM_BATCHES*(0.01))
```

```
[ ]: test_X = t[NUM_TRAIN_BATCH*BATCH_SIZE:
    (NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_X = t[:NUM_TRAIN_BATCH*BATCH_SIZE]
```

```
[ ]: ## Cyclical linear annealing ##
def frange_cycle_linear(n_iter, start=0.0, stop=1.0, n_cycle=4, ratio=0.5):
    L = np.ones(n_iter) * stop
    period = n_iter/n_cycle
```

```

step = (stop-start)/(period*ratio) # linear schedule
for c in range(n_cycle):
    v, i = start, 0
    while v <= stop and (int(i+c*period) < n_iter):
        L[int(i+c*period)] = v
        v += step
        i += 1
return L

```

```

[ ]: def train_smile_vae(vae,train_X, test_X,betas):
    clip = -1
    display_step = 100
    STEPS_PER_EPOCH = train_X.shape[0]//BATCH_SIZE
    TEST_STEPS = test_X.shape[0]//BATCH_SIZE
    LEARNING_RATE = 1e-4
    optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
    ## Index of current annealing
    beta_ind = 0
    for epoch in range(EPOCHS):
        ### Randomize training data
        idxs = np.arange(STEPS_PER_EPOCH)
        np.random.shuffle(idxs)

        for batch in range(STEPS_PER_EPOCH):
            with tf.GradientTape() as tape:
                ## Get relevant batch data
                X_batch = train_X[idxs[batch]*BATCH_SIZE: idxs[batch]*BATCH_SIZE + BATCH_SIZE ]
                ## Get logits of the model
                z_mean, z_log_var, x_decoded = vae(X_batch[:, :-1])
                ## Get loss oppp
                loss_op = vae.vae_loss(labels=X_batch[:, 1:], x_decoded=x_decoded,
                                      z_mean=z_mean,z_log_var=z_log_var,beta=betas[beta_ind])

            ## Apply gradients
            gradients = tape.gradient(loss_op, vae.trainable_variables)
            if clip != -1:
                gradients, _ = tf.clip_by_global_norm(gradients, clip)
            optimizer.apply_gradients(zip(gradients, vae.trainable_variables))

            ### Display information about training ###
            if batch % display_step == 0 or batch == 1:
                ### Get logits for test data
                rand_int = np.random.randint(low=0,high = TEST_STEPS)
                test_batch = test_X[rand_int*BATCH_SIZE: rand_int*BATCH_SIZE + BATCH_SIZE ]

```

```

z_mean, z_log_var, x_decoded = vae(test_batch[:, :-1])
test_loss = vae.vae_loss(labels=test_batch[:, 1:], x_decoded=x_decoded,
                        z_mean=z_mean, z_log_var=z_log_var, beta=✉
                        ↪betas[beta_ind])

## Get KL Loss
kl_loss = vae.get_kl_loss(labels=test_batch[:, 1:], x_decoded=x_decoded,
                           z_mean=z_mean, z_log_var=z_log_var, beta=✉
                           ↪betas[beta_ind])

## Get test logits
pred_test = tf.nn.softmax(x_decoded, axis=-1)
weights = tf.cast(tf.not_equal(test_batch[:, 1:], 0), tf.float32)
nonpad_seq = tf.math.count_nonzero(weights, dtype=tf.dtypes.float32, )
correct_pred_test = tf.equal(tf.argmax(pred_test, -1), test_batch[:, 1:])

## Get test accuracy
accuracy_test = tf.reduce_sum(tf.cast(correct_pred_test, tf.float32))/
↪nonpad_seq

### Print out test accuracy on model
print("Step " + str(batch) + ", Training Loss = " + \
      "{:.3f}".format(loss_op) + ", Test Loss = " + \
      "{:.3f}".format(test_loss) + ", Test KL Loss = " + \
      "{:.3f}".format(kl_loss) + ", Test Accuracy = " + \
      "{:.3f}".format(accuracy_test))

beta_ind += 1
## Save every so often
if (batch) % 3000 == 0:
    vae.save_weights('deep_conv_vae_weights2')

```

[]:

```

betas = frange_cycle_linear(train_X.shape[0]//BATCH_SIZE *EPOCHS)
smile_vae = SMILE_VAE(vocab_size= VOCAB_SIZE, embedding_dim=EMBEDDING_DIM,✉
↪max_len= MAX_LEN,
                      latent_dim = 64, recurrent_dropout =✉
↪DROP_OUT, dropout_rate= DROP_OUT)

```

[]:

```

if TRAIN:
    train_smile_vae(smile_vae, train_X, test_X, betas)
else:
    smile_vae.load_weights('deep_conv_vae_weights2')

```

Train_ImplicitVAE

September 6, 2020

```
[ ]: # Initialize drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

[ ]: # Move to Google Drive
%cd drive
%cd 'My Drive'
%cd 'MSc Stats Dissertation'

[ ]: import numpy as np
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import tensorflow as tf
import tensorflow.keras.backend as K
import tensorflow.keras as keras
import tensorflow.keras.layers as layers
import pandas as pd
import math
import time
import numpy as np
import matplotlib.pyplot as plt

[ ]: BATCH_NORM = True
CONV_ACTIVATION = 'tanh'
CONV_DEPTH = 4
CONV_DIM_DEPTH = 32
CONV_DIM_WIDTH = 16
CONV_D_GF = 1.15875438383
CONV_W_GF = 1.1758149644
HIDDEN_DIM = 100

[ ]: ## Import Necessary Data for training
train_smiles_X = np.load('./vocab/train_smiles_X.npy',allow_pickle=True)
vocab = np.load('./vocab/selfies_vocab.npy',allow_pickle=True)
vocab_index = np.load('./vocab/selfies_vocab_index.npy',allow_pickle=True)
vocab = dict(vocab.ravel()[0])
vocab_index = dict(vocab_index.ravel()[0])
```

```
[ ]: ## Necessary CONSTANTS
BATCH_SIZE = 256
VOCAB_SIZE = len(vocab)
EPOCHS = 40
LEARNING_RATE = 1e-4
Z_X_LEARNING_RATE= 1e-5
DROP_OUT= 0.2
EMBEDDING_DIM = 192 ## Embedding dim of the characters
PAD_LEN = 250 ## Maximum size of a SMILE (100 + BOS, EOS)
MAX_LEN = PAD_LEN -1
DROPOUT = 0.2
LATENT_DIM = 64
HIDDEN_DIM = 256
```

```
[ ]: index = np.where(train_smiles_X == 1)
t = np.split(train_smiles_X,index[0].tolist())
t= t[1:]
t = tf.keras.preprocessing.sequence.
    ↪pad_sequences(t,maxlen=PAD_LEN,padding='post')
NUM_BATCHES = math.floor(len(t)/BATCH_SIZE )
```

```
[ ]: NUM_TRAIN_BATCH = math.floor(NUM_BATCHES*0.99)
NUM_TEST_BATCH = math.floor(NUM_BATCHES*(0.01))
```

```
[ ]: test_X = t[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↪(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_X = t[:NUM_TRAIN_BATCH*BATCH_SIZE]
```

```
[ ]: import math
def softmax_logits_loss_with_pad(labels,logits):
    weights = tf.cast(tf.not_equal(labels, 0), tf.float32)
    nonpad_seq = tf.math.count_nonzero(weights, dtype=tf.dtypes.float32, )
    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=
    ↪logits)
    loss = tf.reduce_sum(loss *weights)
    return loss
```

```
[ ]: ## Cyclical Annealing ###
def frange_cycle_linear(n_iter, start=0.0, stop=1.0, n_cycle=4, ratio=0.5):
    L = np.ones(n_iter) * stop
    period = n_iter/n_cycle
    step = (stop-start)/(period*ratio) # linear schedule
    for c in range(n_cycle):
        v, i = start, 0
        while v <= stop and (int(i+c*period) < n_iter):
            L[int(i+c*period)] = v
            v += step
```

```

    i += 1
    return L

[ ]: def train_smile_ivae(smile_ivae,train_X, test_X,model_type, betas, num_updates):
    clip = -1
    display_step = 100
    STEPS_PER_EPOCH = train_X.shape[0]//BATCH_SIZE
    TEST_STEPS = test_X.shape[0]//BATCH_SIZE

    ## Define optimizers
    optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
    optimizer_xz = tf.keras.optimizers.Adam(learning_rate=Z_X_LEARNING_RATE)
    optimizer_z = tf.keras.optimizers.Adam(learning_rate=Z_X_LEARNING_RATE)

    ## Cyclical Annealing index ##
    beta_ind = 0

    for epoch in range(EPOCHS):
        idxs = np.arange(STEPS_PER_EPOCH)
        np.random.shuffle(idxs)
        for batch in range(STEPS_PER_EPOCH):
            ## Get relevant batch data
            X_batch = train_X[idxs[batch]*BATCH_SIZE: idxs[batch]*BATCH_SIZE + BATCH_SIZE]

            ## Update the auxillary network
            for n in range(num_updates):
                eps = tf.convert_to_tensor(np.random.normal(size=(X_batch.shape[0], LATENT_DIM)), dtype = tf.float32)
                with tf.GradientTape() as kl_xz_tape, tf.GradientTape() as kl_z_tape:

                    ## Get encoding of training data
                    enc, z_x = smile_ivae.encoder(X_batch[:, :-1],eps)
                    kl_xz_vars = smile_ivae.nu_xz.trainable_variables
                    kl_z_vars = smile_ivae.nu_z.trainable_variables

                    ## Get logits of the model
                    kl_xz = smile_ivae.kl_xz_loss(z_x = z_x, enc = enc)
                    kl_z = smile_ivae.kl_z_loss(z_x = z_x)

                    ## KL updates
                    gradients = kl_xz_tape.gradient(kl_xz, kl_xz_vars)
                    if clip != -1:
                        gradients, _ = tf.clip_by_global_norm(gradients, clip)
                    optimizer_xz.apply_gradients(zip(gradients, kl_xz_vars))

                    gradients = kl_z_tape.gradient(kl_z, kl_z_vars)

```

```

if clip != -1:
    gradients, _ = tf.clip_by_global_norm(gradients, clip)
optimizer_z.apply_gradients(zip(gradients, kl_z_vars))

with tf.GradientTape() as tape:
    ## end to end update for implicit model
    x_decoded, enc, z_x = smile_ivae(X_batch[:, :-1])
    vars = smile_ivae.decoder.trainable_variables
    vars.extend(smile_ivae.encoder.trainable_variables)
    loss_op = softmax_logits_loss_with_pad(labels=X_batch[:, 1:], logits=x_decoded)

    ## Model update using auxillary network and softmax
    if model_type == 'mle':
        loss_op = loss_op + (betas[beta_ind]*tf.reduce_sum(smile_ivae.nu_xz(z_x=z_x, enc=enc)))
    else:
        loss_op = loss_op + (betas[beta_ind] *tf.reduce_sum(smile_ivae.nu_z(z_x=z_x)))
    gradients = tape.gradient(loss_op, vars)
    if clip != -1:
        gradients, _ = tf.clip_by_global_norm(gradients, clip)
    optimizer.apply_gradients(zip(gradients, vars))

    ## Display training information
    if batch % display_step == 0 or batch == 1:
        ### Get logits for test data
        rand_int = np.random.randint(low=0, high=TEST_STEPS)
        test_batch = test_X[rand_int*BATCH_SIZE: rand_int*BATCH_SIZE + BATCH_SIZE]
        x_decoded, enc, z_x = smile_ivae(test_batch[:, :-1])
        test_loss = softmax_logits_loss_with_pad(labels=test_batch[:, 1:], logits=x_decoded)

        ## Get testing loss
        if model_type == 'mle':
            test_loss = test_loss +(betas[beta_ind]*tf.reduce_mean(smile_ivae.nu_xz(z_x=z_x, enc=enc)))
        else:
            test_loss = test_loss + (betas[beta_ind] *tf.reduce_mean(smile_ivae.nu_z(z_x=z_x)))

        ## Get testing information for displaying accuracy ##
        pred_test = tf.nn.softmax(x_decoded, axis=-1)
        weights = tf.cast(tf.not_equal(test_batch[:, 1:], 0), tf.float32)
        nonpad_seq = tf.math.count_nonzero(weights, dtype=tf.dtypes.float32, )

```

```

correct_pred_test = tf.equal(tf.argmax(pred_test,-1),test_batch[:,1:])
accuracy_test = tf.reduce_sum(tf.cast(correct_pred_test, tf.float32))/  

→nonpad_seq

## Get regularizing loss for model
if model_type == 'mle':
    kl_loss = betas[beta_ind]*tf.reduce_sum(smile_ivae.nu_xz(z_x = z_x, u  

→enc = enc))
else:
    kl_loss = betas[beta_ind] *tf.reduce_sum(smile_ivae.nu_z(z_x = z_x))
### Print out test accuracy on model
print("Step " + str(batch) + ", Training Loss = " + \
      "{:.3f}".format(loss_op) + ", Test Loss = " + \
      "{:.3f}".format(test_loss) + ", Test Accuracy = " + \
      "{:.3f}".format(accuracy_test) +", Implicit KL value = " + \
      "{:.3f}".format(kl_loss))

beta_ind+=1
## Save every so often
if (batch) % 3000 == 0:
    smile_ivae.save_weights('selfies_ivae_weights')

```

[]:

```

betas = frange_cycle_linear(train_X.shape[0]//BATCH_SIZE *EPOCHS)
smile_ivae = SMILE_IMPLPLICIT_VAE(vocab_size =VOCAB_SIZE,embedding_dim=  

→=EMBEDDING_DIM,
        max_len =MAX_LEN, latent_dim=LATENT_DIM, hidden_dim= HIDDEN_DIM,
        recurrent_dropout =0.2,dropout_rate=0.2,epsilon_std = 1.0)

```

[]:

```

## Train with 3 updates of auxillary network for every end to
## end update
train_smile_vae(smile_ivae,train_X,test_X,'mle_li',betas,3)

```

[]:

```

smile_ivae.summary()

```

Train_IC50Predictions

September 6, 2020

```
[ ]: # Initialize drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

[ ]: # Move to Google Drive
%cd drive
%cd 'My Drive'
%cd 'MSc Stats Dissertation'

[ ]: !wget -c https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
!chmod +x Miniconda3-latest-Linux-x86_64.sh
!time bash ./Miniconda3-latest-Linux-x86_64.sh -b -f -p /usr/local
!time conda install -q -y -c conda-forge rdkit
!pip install selfies
!pip install deepsmiles

[ ]: import sys
import os
sys.path.append('/usr/local/lib/python3.7/site-packages/')

[ ]: import numpy as np
import tensorflow as tf
import tensorflow.keras.backend as K
import tensorflow.keras as keras
import pandas as pd
import math
import tensorflow.keras.layers as layers
from selfies import encoder, decoder
import deepsmiles
import rdkit
import time
import numpy as np
import matplotlib.pyplot as plt

[ ]: ## Import Necessary Data for training
train_smiles_X = np.load('./vocab/train_selfies_X.npy',allow_pickle=True)
vocab =np.load('./vocab/selfies_vocab.npy',allow_pickle=True)
```

```

vocab_index = np.load('./vocab/selfies_vocab_index.npy',allow_pickle=True)
vocab = dict(vocab.ravel()[0])
vocab_index = dict(vocab_index.ravel()[0])
gene_expressions = np.load('./Datasets/ordered_gene_expressions.npy')
ic50 = np.load('./Datasets/ordered_ic50.npy')
smiles_pairs = np.load('./Datasets/ordered_smiles.npy')

```

```

[ ]: ## Neccesary CONSTANTS
BATCH_SIZE = 256
VOCAB_SIZE = len(vocab_index)
EPOCHS = 30
LEARNING_RATE = 1e-4
DROP_OUT= 0.2
EMBEDDING_DIM = 192 ## Embedding dim of the characters
LATENT_DIM = 64
PAD_LEN = 250
MAX_LEN = PAD_LEN -1
DROPOUT = 0.2
TRAIN = True

```

```

[ ]: ## Get randomized chemistry of different smiles strings
## for training
from rdkit import Chem
def randomlabels(mol, N):
    ans = set()
    mol_ex = None
    if mol.find('. ') != -1:
        mol_ex = mol[mol.find('. ') :]
        mol = mol[:mol.find('. ')]
    molt = Chem.MolFromSmiles(mol)
    while len(ans) < N:
        if mol_ex == None:
            ans.add(Chem.MolToSmiles(molt,doRandom=True,canonical=True))
        else:
            ans.add(Chem.MolToSmiles((molt),doRandom=True,canonical=True)+mol_ex)
    return sorted(list(ans))

```

```

[ ]: ## converts smiles to selfies smiles
def get_smiles_from_selfies(selfies_list):
    smiles = []
    for selfie in selfies_list:
        try:
            smile = decoder(selfie)
        except:
            smile = None
        smiles.append(smile)
    return smiles

```

```
[ ]: def integer_encode_selfies(selfies,vocab_dict):
    selfies_enc = []
    for char in selfies:
        selfies_enc.append(vocab_dict[char])
    return selfies_enc

[ ]: ## Splits the selfies <molecule> into a list of character strings.
def split_selfie(molecule):
    return re.findall(r'\[.*?\]|\.\.', molecule)

[ ]: ## Takes processed selfies smiles and returns the tokenized
## versions of the selfies
def tokenize_selfies(selfies):
    char_list = split_selfie(selfies)
    tokenized= []
    tokenized.append('<BOS>')
    i = 0
    while i < len(char_list):
        char = char_list[i]
        tokenized.append(char)
        i = i+1
    tokenized.append('<EOS>')
    return tokenized

[ ]: import re
## replace Br and Cl with single letters
def replace_halogens(string):
    br = re.compile('Br')
    cl = re.compile('Cl')
    string = br.sub('R', string)
    string = cl.sub('L', string)
    return string

[ ]: ### Properly scale data
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(ic50.reshape(-1,1))
ic50 = scaler.transform(ic50.reshape(-1,1))
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(gene_expressions)
gene_expressions = scaler.transform(gene_expressions)

[ ]: new_gene_expressions = []
new_ic50 = []
new_smile_pairs = []
```

```
[ ]: ## Get training data for guided training
for i in range(len(smiles_pairs)):
    if i % 10000 == 0:
        print(i)
    random_mols = randomlabels(smiles_pairs[i], 8)
    if SELFIES:
        for mols in random_mols:
            new_smile_pairs.append(encoder(mols))
    else:
        new_smile_pairs.extend(random_mols)
    for j in range(len(random_mols)):
        new_gene_expressions.append(gene_expressions[i])
        new_ic50.append(ic50[i])
gene_expressions= new_gene_expressions
ic50 = new_ic50
smiles_pairs = new_smile_pairs
```

```
[ ]: smile_pair_tokens = []
for smiles in smiles_pairs:
    if SELFIES:
        smile_pair_tokens.append(tokenize_selfies(smiles))
    else:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(smiles))
smile_pair_tokens = np.array(smile_pair_tokens)
```

```
[ ]: smiles_ordered = []
for smiles in smile_pair_tokens:
    if SELFIES:
        smiles_ordered.append(integer_encode_selfies(smiles, vocab))
    else:
        smiles_ordered.append(integer_encode(smiles, vocab))
smiles_ordered = np.array(smiles_ordered)
```

```
[ ]: t = tf.keras.preprocessing.sequence.
    ↪pad_sequences(smiles_ordered, maxlen=PAD_LEN, padding='post')
NUM_BATCHES = math.floor(len(t)/BATCH_SIZE )
```

```
[ ]: NUM_TRAIN_BATCH = math.floor(NUM_BATCHES*0.99)
NUM_TEST_BATCH = math.floor(NUM_BATCHES*(0.01))
```

```
[ ]: test_smiles = t[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↪(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_smiles = t[:NUM_TRAIN_BATCH*BATCH_SIZE]

test_genes = gene_expressions[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↪(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
```

```

train_genes = gene_expressions[:NUM_TRAIN_BATCH*BATCH_SIZE]

test_ic50 = ic50[NUM_TRAIN_BATCH*BATCH_SIZE:
                  ↵(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_ic50 = ic50[:NUM_TRAIN_BATCH*BATCH_SIZE]

[ ]: import GANS.ic50vae as conv_smiles_vae
smile_vae = conv_smiles_vae.SMILE_VAE(vocab_size=_
                                      ↵VOCAB_SIZE, embedding_dim=EMBEDDING_DIM, max_len= MAX_LEN,
                                      latent_dim = LATENT_DIM, recurrent_dropout =_
                                      ↵DROP_OUT, dropout_rate= DROP_OUT)
if SELFIES:
    smile_vae.load_weights('ic50g_selfies_conv_vae_weights')
else:
    smile_vae.load_weights('smiles_conv_vae_weights2')

[ ]: mca_ic50 = ic50mca.IC50_MCA(vocab_size=VOCAB_SIZE,
                                  embedding_dim =EMBEDDING_DIM, num_genes =2128,
                                  hidden_dim = HIDDEN_DIM, max_len = MAX_LEN,
                                  latent_dim = LATENT_DIM)

[ ]: def train_smile_gene_ca(mca_ic50,smile_vae,
                           train_smiles,train_genes,train_ic50,
                           test_smiles, test_genes,test_ic50):
    clip = -1
    display_step = 100
    STEPS_PER_EPOCH = train_smiles.shape[0]//BATCH_SIZE
    TEST_STEPS = test_smiles.shape[0]//BATCH_SIZE
    optimizer = tf.keras.optimizers.Adam(learning_rate=5e-6)

    for epoch in range(EPOCHS):
        ## Randomize the training process
        idxs = np.arange(STEPS_PER_EPOCH)
        np.random.shuffle(idxs)
        total_train_loss = 0
        for batch in range(STEPS_PER_EPOCH):
            with tf.GradientTape() as tape:

                ## Get relevant batch data
                train_smiles_batch = np.array(train_smiles[idxs[batch]*BATCH_SIZE:_]
                                              ↵idxs[batch]*BATCH_SIZE + BATCH_SIZE])
                train_gene_batch = np.array(train_genes[idxs[batch]*BATCH_SIZE:_]
                                              ↵idxs[batch]*BATCH_SIZE + BATCH_SIZE])
                train_ic50_batch = np.array(train_ic50[idxs[batch]*BATCH_SIZE:_]
                                              ↵idxs[batch]*BATCH_SIZE + BATCH_SIZE])
                ## Get logits of the model

```

```

h, z_mean,z_log_var = smile_vae.encoder(train_smiles_batch[:, :-1])
z = tf.keras.layers.Lambda(smile_vae.encoder.sample,
                           output_shape =(LATENT_DIM,))([z_mean,z_log_var])

## Get ic50 prediction
ic50_pred = mca_ic50(encoded_smiles = z_mean, genes = train_gene_batch)
loss_op = tf.reduce_sum(tf.keras.losses.MSE(train_ic50_batch,ic50_pred))
total_train_loss += loss_op

## Apply gradients
gradients = tape.gradient(loss_op, mca_ic50.trainable_variables)
if clip != -1:
    gradients, _ = tf.clip_by_global_norm(gradients, clip)
optimizer.apply_gradients(zip(gradients, mca_ic50.trainable_variables))

## Display training loss information ##
if batch % display_step == 0 or batch == 1:

    ### Get logits for test data
    rand_int =np.random.randint(low=0,high = TEST_STEPS)
    test_smiles_batch = np.array(test_smiles[rand_int*BATCH_SIZE:�
    ↪rand_int*BATCH_SIZE + BATCH_SIZE])
    test_gene_batch = np.array(test_genes[rand_int*BATCH_SIZE:�
    ↪rand_int*BATCH_SIZE + BATCH_SIZE])
    test_ic50_batch = np.array(test_ic50[rand_int*BATCH_SIZE:�
    ↪rand_int*BATCH_SIZE + BATCH_SIZE])

    ### Encode data using encoder
    h, z_mean,z_log_var = smile_vae.encoder(test_smiles_batch[:, :-1])
    z = tf.keras.layers.Lambda(smile_vae.encoder.sample,
                               output_shape =(LATENT_DIM,))([z_mean,z_log_var])

    ## Get prediction
    ic50_pred_test = mca_ic50(encoded_smiles = z_mean,
                               genes = test_gene_batch,training=False)

    ### Print out test accuracy on model
    test_loss = tf.reduce_sum(tf.keras.losses.MSE(test_ic50_batch,
                                                 ic50_pred_test))

    print("Step " + str(batch) + ", Training Loss = " + \
          "{:.3f}".format(tf.reduce_mean(loss_op)) + ", Test Loss = " + \
          "{:.3f}".format(tf.reduce_mean(test_loss)))
    if (batch) == STEPS_PER_EPOCH-1:
        print("TOTAL TRAINING LOSS " + "{:.3f}".format(tf.
        ↪reduce_sum(total_train_loss)))
        if (batch) % 3000 == 0:

```

```
mca_ic50.save_weights('ic50network_selfies_ic50g')

[ ]: if TRAIN:
    train_smile_gene_ca(mca_ic50,smile_vae,
                         train_smiles,train_genes,train_ic50,
                         test_smiles, test_genes,test_ic50)
else:
    mca_ic50.load_weights('ic50network_selfies_ic50g')

[ ]: mca_ic50.summary()
```

Ensmeble_PredTox

September 6, 2020

```
[ ]: # Initialize drive/
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

[ ]: # Move to Google Drive
%cd drive
%cd 'My Drive'
%cd 'MSc Stats Dissertation'

[ ]: ## Install necessary libraries
!pip install deepsmiles
!pip install selfies
!wget -c https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
!chmod +x Miniconda3-latest-Linux-x86_64.sh
!time bash ./Miniconda3-latest-Linux-x86_64.sh -b -f -p /usr/local
!time conda install -q -y -c conda-forge rdkit

[ ]: ## Go to correct place in drive to allow us
## to import libraries
import sys
import os
sys.path.append('/usr/local/lib/python3.7/site-packages/')

[ ]: ## Import Necessary Libraries
import numpy as np
import tensorflow as tf
import tensorflow.keras.backend as K
import tensorflow.keras as keras
import pandas as pd
import math
import tensorflow.keras.layers as layers
import time
import numpy as np
import matplotlib.pyplot as plt
import lightgbm as lgb
from sklearn.metrics import matthews_corrcoef
import deepsmiles
```

```

from selfies import encoder, decoder
import rdkit
import Utils.generate_utils as generate_utils
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis, QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from imblearn.under_sampling import RandomUnderSampler

```

[]: *## Converter to convert SMILES to Deep SMILES*
 converter = deepsmiles.Converter(rings = True, branches = True)

[]: *## Set to correct float type for consistency with training*
 tf.keras.backend.set_floatx('float32')

[]: *## Import data files*
 SELFIES = False
 DEEP = False
 train_smiles_path = './Datasets/train_Tox_data.smi'
 test_smiles_path = './Datasets/AID_1189_datatable_all.csv'
 actual_test_smiles_path = './Datasets/3643044675069416146.txt'
 if SELFIES:
 vocab = np.load('./vocab/selfies_vocab.npy', allow_pickle=True)
 vocab_index = np.load('./vocab/selfies_vocab_index.npy', allow_pickle=True)
 elif DEEP:
 vocab = np.load('./vocab/deep_vocab.npy', allow_pickle=True)
 vocab_index = np.load('./vocab/deep_vocab_index.npy', allow_pickle=True)
 else:
 vocab = np.load('./vocab/vocab.npy', allow_pickle=True)
 vocab_index = np.load('./vocab/vocab_index.npy', allow_pickle=True)
 vocab = dict(vocab.ravel()[0])
 vocab_index = dict(vocab_index.ravel()[0])
 smiles_train = pd.read_csv(train_smiles_path, delimiter='\t', header=None)
 MIN = 3027

[]: *## Load in file and process it for later predictions*
 smiles_test = pd.read_csv(test_smiles_path, delimiter=',')

```

actual_test_smiles = pd.read_csv(actual_test_smiles_path,delimiter='\t',header=None)
train = []
test = []
train_Y = []
test_Y = []
for index in range(len(smiles_train[1])):
    if smiles_train[1][index][:2] == 'DB':
        test.append(index)
        test_Y.append(0)
    elif smiles_train[1][index][:1] == 'D':
        train.append(index)
        train_Y.append(0)
    elif smiles_train[1][index][:2] == 'T3':
        test.append(index)
        test_Y.append(1)
    else:
        train.append(index)
        train_Y.append(1)

train_X = smiles_train[0][train]
test_X = smiles_train[0][test]

```

```

[ ]: ## Necessary CONSTANTS
BATCH_SIZE = 64
VOCAB_SIZE = len(vocab)
EPOCHS = 10
LEARNING_RATE = 0.000312087049936
if SELFIES or DEEP:
    PAD_LEN = 250
else:
    PAD_LEN = 160 ## Maximum size of a SMILE (100 + BOS, EOS)
    print('HERE')
MAX_LEN = PAD_LEN
DROP_OUT= 0.2
EMBEDDING_DIM = 192 ## Embedding dim of the characters
HIDDEN_DIM = 256
DROPOUT = 0.2
TRAIN = False
LATENT_DIM = 64
TRANSFORMER_DECODE = True

```

```

[ ]: ## Integer encode for selfies
def integer_encode_selfies(selfies,vocab_dict):
    selfies_enc = []
    for char in selfies:
        try:

```

```

        selfies_enc.append(vocab_dict[char])
    except:
        return None
    return selfies_enc

```

```
[ ]: ## Splits the selfies <molecule> into a list of character strings.
def split_selfie(molecule):
    return re.findall(r'\[.*?\]|\.\.', molecule)

## Takes processed selfies smiles and returns the tokenized
## versions of the selfies
def tokenize_selfies(selfies):
    char_list = split_selfie(selfies)
    tokenized= []
    tokenized.append('<BOS>')
    i = 0
    while i < len(char_list):
        char = char_list[i]
        tokenized.append(char)
        i = i+1
    tokenized.append('<EOS>')
    return tokenized
```

```
[ ]: import re
## replace Br and Cl with single letters
def replace_halogens(string):
    br = re.compile('Br')
    cl = re.compile('Cl')
    string = br.sub('R', string)
    string = cl.sub('L', string)
    return string
```

```
[ ]: ## Takes processed smiles/deep smiles and returns the tokenized
## versions of the smiles or deep semiles
## Note: Run replace halogens and replace percentages
## before running this method
def tokenize_smiles(smiles):
    char_list = list(smiles)
    tokenized= []
    tokenized.append('<BOS>')
    i = 0
    while i < len(char_list):
        char = char_list[i]
        tokenized.append(char)
        i= i+1
    tokenized.append('<EOS>')
    return tokenized
```

```
[ ]: ## Integer encode fore SMILES and DeepSMILES
def integer_encode(smiles,vocab_dict):
    smiles_enc = []
    for char in smiles:
        if char in vocab:
            smiles_enc.append(vocab_dict[char])
        else:
            return None
    return smiles_enc
```

```
[ ]: ## Process for later prediction
smile_pair_tokens = []
indexes = []
index = 0
for smiles in train_X:
    if SELFIES:
        smiles = encoder(smiles)
        if smiles is not None:
            indexes.append(index)
            smile_pair_tokens.append(tokenize_selfies(smiles))
    elif DEEP:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(converter.encode(smiles)))
    else:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(smiles))
    index = index+1
smile_pair_tokens = np.array(smile_pair_tokens)
if SELFIES:
    train_Y = np.array(train_Y)[indexes]

smiles_ordered = tf.keras.preprocessing.sequence.
    pad_sequences(smiles_ordered,maxlen = PAD_LEN,padding='post')
tox_smiles = np.array(smiles_ordered)
```

```
[ ]: ## Prrocess for later prediction
smiles_ordered = []
indexes = []
index = 0
for smiles in smile_pair_tokens:
    if SELFIES:
        encoded_smile = integer_encode_selfies(smiles,vocab)
        if encoded_smile is not None:
            smiles_ordered.append(encoded_smile)
        else:
            smiles_ordered.append(None)
```

```

else:
    smiles_ordered.append(integer_encode(smiles,vocab))
l=[i for i,v in enumerate(smiles_ordered) if v != None ]
smiles_ordered = np.array(smiles_ordered)[l]
train_Y = np.array(train_Y)[l]
l=[i for i,v in enumerate(smiles_ordered) if len(v) <MAX_LEN]
smiles_ordered = np.array(smiles_ordered)[l]
train_Y = train_Y[l]

smile_pair_tokens = []
indexes = []
index = 0
for smiles in test_X:
    if SELFIES:
        smiles = encoder(smiles)
        if smiles is not None:
            indexes.append(index)
            smile_pair_tokens.append(tokenize_selfies(smiles))
    elif DEEP:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(converter.encode(smiles)))
    else:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(smiles))
    index = index+1
smile_pair_tokens = np.array(smile_pair_tokens)
if SELFIES:
    test_Y = np.array(test_Y)[indexes]

smiles_ordered = []
for smiles in smile_pair_tokens:
    if SELFIES:
        encoded_smile = integer_encode_selfies(smiles,vocab)
        smiles_ordered.append(encoded_smile)
    else:
        smiles_ordered.append(integer_encode(smiles,vocab))
l=[i for i,v in enumerate(smiles_ordered) if v != None ]
smiles_ordered = np.array(smiles_ordered)[l]
test_Y = np.array(test_Y)[l]
l=[i for i,v in enumerate(smiles_ordered) if len(v) <MAX_LEN]
cancer_smiles = np.array(smiles_ordered)[l]
test_Y = test_Y[l]

```

```
[ ]: import GANS.renewed_smiles_vae as conv_smiles_vae
import GANS.ic50vae as ic50vae

## Import in the VAE so that embeddings of different compounds can be calculated
if IC50:
    smile_vae = ic50vae.SMILE_VAE(vocab_size=_
        →VOCAB_SIZE,embedding_dim=EMBEDDING_DIM, max_len= MAX_LEN,
                    latent_dim = LATENT_DIM, recurrent_dropout =_
        →DROP_OUT,dropout_rate= DROP_OUT)
    if DEEP:
        print('IC50 DEEP')
        smile_vae.load_weights('ic50g_deep_conv_vae_weights')
    elif SELFIES:
        print('IC50 SELFIES')
        smile_vae.load_weights('ic50g_selfies_conv_vae_weights')
    else:
        print('IC50 NORMAL')
        smile_vae.load_weights('ic50g_smiles_conv_vae_weights')
else:
    smile_vae = conv_smiles_vae.SMILE_VAE(vocab_size=_
        →VOCAB_SIZE,embedding_dim=EMBEDDING_DIM, max_len= MAX_LEN,
                    latent_dim = LATENT_DIM, recurrent_dropout =_
        →DROP_OUT,dropout_rate= DROP_OUT)
    if DEEP:
        print('DEEP')
        smile_vae.load_weights('deep_conv_vae_weights2')
    elif SELFIES:
        print('SELFIES')
        smile_vae.load_weights('selfies_conv_vae_weights2')
    else:
        print('NORMAL')
        smile_vae.load_weights('smiles_conv_vae_weights2')
```

```
[ ]: ## Get the latents representations for the training data
train_latents = []
index = 0
for smile in smiles_ordered:
    train_latents.append(smile_vae.encoder(smile.reshape(1,MAX_LEN))[1])
    if index %1000 == 0:
        print(index)
    index+=1
```

```
[ ]: ## Get test representations for the test data
cancer_smiles = tf.keras.preprocessing.sequence.
    →pad_sequences(cancer_smiles,maxlen = PAD_LEN,padding='post')
cancer_smiles = np.array(cancer_smiles)
```

```

test_latents = []
index = 0
for smile in cancer_smiles:
    test_latents.append(smile_vae.encoder(smile.reshape(1,MAX_LEN))[1])
    if index %100 == 0:
        print(index)
    index+=1
test_latents = np.array(test_latents)

```

```

[ ]: def train_model(model,train_X,train_Y,test_X,test_Y):
    model.fit(train_X,train_Y.ravel())
    predictions = model.predict(test_X)
    probs = model.predict_proba(test_X)
    print(classification_report(test_Y, predictions))
    print(confusion_matrix(test_Y, predictions))
    print(accuracy_score(test_Y, predictions))
    print(matthews_corrcoef(test_Y, predictions))
    return predictions, probs

```

```

[ ]: def test_on_model(model,test_X,test_Y):
    predictions = model.predict(test_X)
    probs = model.predict_proba(test_X)
    print(classification_report(test_Y, predictions))
    print(confusion_matrix(test_Y, predictions))
    print(accuracy_score(test_Y, predictions))
    print(matthews_corrcoef(test_Y, predictions))
    return predictions, probs

```

```

[ ]: def get_mcc_curve(y_true, predictions_prob):
    cutoffs = np.arange(0,1,1e-4)
    mccs = []
    for cutoff in cutoffs:
        predictions = labels = (predictions_prob > cutoff).astype(np.int)
        mccs.append(matthews_corrcoef(y_true, predictions))
    return cutoffs, mccs

```

```

[ ]: import matplotlib
## plot ROC curve
def plot_roc_curve(y_true, y_probs, title):
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    fpr, tpr, _ = roc_curve(y_true, y_probs, pos_label=1)
    roc_auc = auc(fpr, tpr)
    matplotlib.rcParams['font', size=20]
    fig = plt.figure(figsize=(8, 8))
    lw = 2

```

```

plt.plot(fpr, tpr, color='black',
         lw=lw, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.legend(loc="lower right")
fig.tight_layout()
fig.savefig("./Figures/" + title + ".pdf", bbox_inches='tight')
plt.show()

## plot MCC curve
def plot_mcc_curve(y_true, y_probs,title ):
    cuttofs, mccs = get_mcc_curve(y_true,y_probs)
    matplotlib.rc('font', size=20)
    fig = plt.figure(figsize=(8, 8))
    lw = 2
    plt.plot(cuttofs, mccs, color='black',
              lw=lw)
    #plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 0.41])
    plt.xlabel('Toxicity Probability Cutoff')
    plt.ylabel('MCC')
    #plt.legend(loc="lower right")
    fig.tight_layout()
    fig.savefig("./Figures/" + title + ".pdf", bbox_inches='tight')
    plt.show()

## plot Confusion Matrix
def plot_confusion_matrix(y_true, y_pred, classes, title,
                           normalize=False, cmap=plt.cm.Blues):
    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    # classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    matplotlib.rc('font', size=20)
    fig = plt.figure(figsize=(12, 12))
    ax = fig.add_subplot(111)

```

```

im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
ax.figure.colorbar(im, ax=ax)
# We want to show all ticks...
ax.set(xticks=np.arange(cm.shape[1]),
       yticks=np.arange(cm.shape[0]),
       # ... and label them with the respective list entries
       xticklabels=classes, yticklabels=classes,
       ylabel='True label',
       xlabel='Predicted label')
_ = plt.xlabel("Predicted Labels", fontsize=18)
_ = plt.ylabel("True label", fontsize=18)

plt.rc('xtick', labelsize=14)
plt.rc('ytick', labelsize=14)
# Rotate the tick labels and set their alignment.
plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

# Loop over data dimensions and create text annotations.
fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        ax.text(j, i, format(cm[i, j], fmt),
                ha="center", va="center", fontsize=14,
                color="white" if cm[i, j] > thresh else "black")
fig.tight_layout()
fig.savefig("./Figures/" + title + ".pdf", bbox_inches='tight')
return ax

```

```
[ ]: train_latents = np.array(train_latents).reshape(-1,64)#
test_latents = np.array(test_latents).reshape(-1,64)
```

```
[ ]: ## Parameters for performing grid searches
params = {
    'boosting_type': 'dart',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'learning_rate': 0.05,
    'feature_fraction': 0.85,
    'bagging_fraction': 0.8,
    'bagging_freq': 5,
    'verbose': 0,
}
```

```
param_grid ={ 'boosting_type': ['dart','lgbm'],
```

```

'objective': ['binary'],
'n_estimators': [1200,1400,1600,1800],
'learning_rate': [0.1,0.05],
'feature_fraction': [0.9,1.0],
'num_leaves' :[31,63,127,255,511,800],
'lambda_l2 ':[0.0,0.1,0.5,1,5]
}

```

```

[ ]: # make an ensemble prediction for classification
def ensemble_probabilities(members, testX):
    # make predictions
    yhats = [model.predict_proba(testX)[:,1] for model in members]
    yhats = np.array(yhats)
    # sum across ensemble members
    summed = np.sum(yhats, axis=0)
    # argmax across classes
    result = summed/len(members)
    return np.reshape(result,(len(result),1))
def ensemble_predictions(members, testX):
    # make predictions
    yhats = [model.predict(testX) for model in members]
    yhats = np.array(yhats)
    # sum across ensemble members
    summed = np.sum(yhats, axis=0)
    # argmax across classes
    result = summed/len(members)
    return np.reshape(result,(len(result),1))

```

```

[ ]: from sklearn import metrics

def TPR(y_true, y_pred):
    # counts the number of true positives (y_true = 1, y_pred = 1)
    tp = list((y_true == 1) & (y_pred == 1)).count(True)
    n = list((y_true == 1)).count(True)
    return tp/n
def FNR(y_true, y_pred):
    # counts the number of false negatives (y_true = 1, y_pred = 0)
    fn = list((y_true == 1) & (y_pred == 0)).count(True)
    p = list(y_true == 1).count(True)
    return fn/p
def FPR(y_true, y_pred):
    # counts the number of false positives (y_true = 0, y_pred = 1)
    fp = list((y_true == 0) & (y_pred == 1)).count(True)
    n = list(y_true == 0).count(True)
    return fp/n
def TNR(y_true, y_pred):
    # counts the number of true negatives (y_true = 0, y_pred = 0)

```

```

tn= list((y_true == 0) & (y_pred == 0)).count(True)
n = list(y_true == 0).count(True)
return tn/n

```

[]: ## Found parameters for the DART and LGBM models

```

params_lgbm = {
    'boosting_type': 'gbdt',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'learning_rate': 0.05,
    'feature_fraction': 1.0,
    'num_leaves':63,
    'verbose': 1,
    'min_data_in_leaf':10,
    'n_estimators' : 2000,
}

```

```

params_dart = {
    'boosting_type': 'dart',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'learning_rate': 0.05,
    'feature_fraction': 1.0,
    'num_leaves':127,
    'verbose': 1,
    'min_data_in_leaf':10,
    'n_estimators' : 2000,
}

```

[]: from sklearn.ensemble import ExtraTreesClassifier

```

param_grid ={ 'max_depth': [10, 20, 30, 40, 50,60,70,80,90,100],
              'n_estimators': [100,200, 400, 600, 800, 1000, 1200]}

```

[]: import tensorflow.keras as keras

```

def get_nnmodel():
    my_init = keras.initializers.glorot_uniform(seed=1)
    model = keras.models.Sequential()
    model.add(keras.layers.Dense(units=4096, input_dim=64,
                                activation='relu', kernel_initializer=my_init))
    model.add(keras.layers.Dropout(0.7))
    model.add(keras.layers.Dense(units=2048, activation='relu',
                                kernel_initializer=my_init))
    model.add(keras.layers.Dropout(0.5))
    model.add(keras.layers.Dense(units=1024, activation='relu',
                                kernel_initializer=my_init))

```

```

model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=512, activation='relu',
    kernel_initializer=my_init))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=256, activation='relu',
    kernel_initializer=my_init))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=512, activation='relu',
    kernel_initializer=my_init))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=1024, activation='relu',
    kernel_initializer=my_init))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(units=1, activation='sigmoid',
    kernel_initializer=my_init))
simple_sgd = keras.optimizers.SGD(lr=0.1)
model.compile(loss='binary_crossentropy',
    optimizer=simple_sgd, metrics=['accuracy'])
return model

```

```
[ ]: def fit_QDAmodel(train_X, train_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
    ↴},random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y.ravel())
    y_rs = np.array(y_rs)
    x_rs = np.array(x_rs)
    QDA = QuadraticDiscriminantAnalysis()
    QDA.fit(x_rs,y_rs)
    return QDA
```

```
[ ]: def fit_LGBmodel(train_X, train_Y,test_X,test_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
    ↴},random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y)
    lgb_train = lgb.Dataset(x_rs, y_rs)
    lgb_eval = lgb.Dataset(test_X, test_Y.flatten(), reference=lgb_train)
    model = lgb.train(params_lgbm,
                      lgb_train,
                      num_boost_round=2000,
                      valid_sets=lgb_eval,
                      early_stopping_rounds=100,
                      verbose_eval=True)
    return model
```

```
[ ]: def fit_DARTmodel(train_X, train_Y,test_X,test_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
→},random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y)
    lgb_train = lgb.Dataset(x_rs, y_rs)
    lgb_eval = lgb.Dataset(test_X, test_Y.flatten(), reference=lgb_train)
    model = lgb.train(params_dart,
                      lgb_train,
                      num_boost_round=100,
                      valid_sets=lgb_eval,
                      early_stopping_rounds=10,
                      verbose_eval=True)
    return model
```

```
[ ]: def fit_SVMmodel(train_X, train_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
→},random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y.ravel())
    y_rs = np.array(y_rs)
    x_rs = np.array(x_rs)
    SVM = SVC(probability = True)
    SVM.fit(x_rs,y_rs)
    return SVM
```

```
[ ]: def fit_LDAmodeL(train_X, train_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
→},random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y.ravel())
    y_rs = np.array(y_rs)
    x_rs = np.array(x_rs)
    LDA = LinearDiscriminantAnalysis()
    LDA.fit(x_rs,y_rs)
    return LDA
```

```
[ ]: from imblearn.over_sampling import RandomOverSampler
def fit_nnmodel(train_X, train_Y,test_X,test_Y):
    global random_state;
    random_state= random_state +1
    sampler = RandomUnderSampler(ratio={0: MIN    , 1:MIN
→},random_state=random_state)
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y)
```

```

model = get_nnmodel()
model.fit(x_rs,
           y_rs,
           batch_size=128,
           shuffle=True,
           validation_data=(test_X,test_Y),
           epochs=10,
           verbose=0)
return model

```

```

[ ]: from sklearn.ensemble import ExtraTreesClassifier
def fit_etmodel(train_X, train_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN , 1:MIN
                                         },random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y)
    etc = ExtraTreesClassifier(n_estimators=1000,max_depth=50)
    etc.fit(train_X,train_Y)
    return etc

```

```

[ ]: def fit_rfmodel(train_X, train_Y):
    global random_state;
    sampler = RandomUnderSampler(ratio={0: MIN , 1:MIN
                                         },random_state=random_state)
    random_state= random_state +1
    x_rs, y_rs = sampler.fit_sample(train_X, train_Y)
    random_forest = RandomForestClassifier(n_estimators=1000,max_depth=30)
    random_forest.fit(train_X,train_Y)
    return random_forest

```

```

[ ]: n_members = 1
random_state= 0
members_rf = [fit_rfmodel(train_latents, train_Y) for _ in range(n_members)]

```

```

[ ]: predictions_rf = ensemble_predictions(members_rf,test_latents)

```

```

[ ]: n_members = 10
random_state= 0
members_nn = [fit_nnmodel(train_latents, train_Y,test_latents,test_Y) for _ in
               range(n_members)]
predictions_nn = ensemble_predictions(members_nn,test_latents)

```

```

[ ]: n_members = 10
random_state= 0
members_lda = [fit_LDAmodel(train_latents, train_Y) for _ in range(n_members)]
predictions_lda = ensemble_predictions(members_lda,test_latents)

```

```
[ ]: n_members = 10
random_state= 0
members_qda = [fit_QDAmodel(train_latents, train_Y) for _ in range(n_members)]
predictions_qda = ensemble_predictions(members_qda,test_latents)
```

```
[ ]: n_members = 10
random_state = 0
members_dart =[fit_DARTmodel(train_latents, train_Y,test_latents,test_Y) for _ in
               range(n_members)]
predictions_dart = ensemble_predictions(members_dart,test_latents)
```

```
[ ]: n_members = 10
random_state = 0
members_lgbm =[fit_LGBmodel(train_latents, train_Y,test_latents,test_Y) for _ in
                range(n_members)]
predictions_lgbm = ensemble_predictions(members_lgbm,test_latents)
```

```
[ ]: n_members = 10
random_state= 0
members_svm = [fit_SVMmodel(train_latents, train_Y) for _ in range(n_members)]
predictions_svm= ensemble_predictions(members_svm,test_latents)
```

```
[ ]: n_members = 1
random_state= 0
members_etc = [fit_etmodel(train_latents, train_Y) for _ in range(n_members)]
predictions_etc = ensemble_predictions(members_etc,test_latents)
```

```
[ ]: all_models = []
all_models.append(members_lgbm)
all_models.append(members_dart)
all_models.append(members_qda)
all_models.append(members_rf)
all_models.append(members_nn)
all_models.append(members_lda)
all_models.append(members_svm)
all_models.append(members_etc)
```

```
[ ]: all_probs = []
all_probs.append(predictions_lgbm)
all_probs.append(predictions_dart)
all_probs.append(predictions_qda)
all_probs.append(predictions_rf)
all_probs.append(predictions_nn)
#ll_probs.append(predictions_lda)
all_probs.append(predictions_svm)
all_probs.append(predictions_etc)
```

```
[ ]: # calculated a weighted sum of predictions
def weighted_sum(weights, yhats):
    rows = list()
    for j in range(yhats.shape[1]):
        # enumerate values
        row = list()
        for k in range(yhats.shape[2]):
            # enumerate members
            value = 0.0
            for i in range(yhats.shape[0]):
                value += weights[i] * yhats[i,j,k]
            row.append(value)
        rows.append(row)
    return array(rows)

# make an ensemble prediction for binary classification
def ensemble_predictions2(weights,all_probs):
    # make predictions
    #yhats = [model.predict(testX) for model in members]
    yhats = all_probs
    yhats = np.array(yhats)
    # weighted sum across ensemble members
    summed = np.tensordot(yhats, weights, axes=((0),(0)))
    # argmax across classes
    result = summed
    return result

# evaluate a specific number of members in an ensemble
def evaluate_ensemble(weights,test_Y,all_probs):
    # make prediction
    yhat = ensemble_predictions2(weights,all_probs)
    labels = (yhat > 0.5).astype(np.int)
    fpr, tpr, _ = roc_curve(np.array(test_Y).flatten(), np.array(yhat).
    flatten(), pos_label=1)
    roc_auc = auc(fpr, tpr)
    # calculate accuracy
    #r2(np.array(test_Y).flatten(),np.array(yhat).flatten())
    return roc_auc

def normalize(weights):
    # calculate l1 vector norm
    result = norm(weights, 1)
    # check for a vector of all zeros
    if result == 0.0:
        return weights
    # return normalized vector (unit norm)
```

```

    return weights / result

# grid search weights
def grid_search(test_Y,all_probs):
    # define weights to consider
    w = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
    best_score, best_weights = 0.0, None
    # iterate all possible combinations (cartesian product)
    for weights in product(w, repeat=len(all_probs)):
        # skip if all weights are equal
        if len(set(weights)) == 1:
            continue
        # hack, normalize weight vector
        weights = normalize(weights)
        # evaluate weights
        score = evaluate_ensemble(weights,test_Y,all_probs)
        if score > best_score:
            best_score, best_weights = score, weights
            print('>%s %.3f' % (best_weights, best_score))
    return list(best_weights)

# grid search for coefficients in a weighted average ensemble for the blobs
# problem
from sklearn.datasets import make_blobs
from sklearn.metrics import accuracy_score
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from matplotlib import pyplot
from numpy import mean
from numpy import std
from numpy import array
from numpy import argmax
from numpy import tensordot
from numpy.linalg import norm
from itertools import product
# grid search weights
weights = grid_search(test_Y,all_probs)
score = evaluate_ensemble(weights,test_Y,all_probs)
print('Grid Search Weights: %s, Score: %.3f' % (weights, score))

```

Optimization_EvaluateSmileModel (1)

September 6, 2020

```
[ ]: # Initialize drive
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

[ ]: # Move to Google Drive
%cd drive
%cd 'My Drive'
%cd 'MSc Stats Dissertation'

[ ]: ## Install necessary additional libraries
!pip install deepsmiles
!pip install selfies==0.2.4
!pip install GPyOpt
!wget -c https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
!chmod +x Miniconda3-latest-Linux-x86_64.sh
!time bash ./Miniconda3-latest-Linux-x86_64.sh -b -f -p /usr/local
!time conda install -q -y -c conda-forge rdkit
!pip install guacamol

[ ]: ## Go to correct place in drive to allow us
## to import libraries
import sys
import os
sys.path.append('/usr/local/lib/python3.7/site-packages/')

[ ]: ## Import Necessary lIbraries
import numpy as np
import deepsmiles
import tensorflow as tf
import tensorflow as tf
import tensorflow.keras.backend as K
import tensorflow.keras as keras
import pandas as pd
import math
import tensorflow.keras.layers as layers
import rdkit
import Utils.generate_utils as generate_utils
```

```

import time
import numpy as np
import matplotlib.pyplot as plt
from guacamol.distribution_learning_benchmark import ValidityBenchmark, \
    UniquenessBenchmark, NoveltyBenchmark, KLDivBenchmark
from selfies import encoder, decoder
import GPyOpt
from GPyOpt.methods import BayesianOptimization
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
from numpy.random import multivariate_normal
import GPyOpt
from GPyOpt.methods import BayesianOptimization
import rdkit
import rdkit.Chem.Descriptors as Descriptors
import Utils.sascorer as sascorer

```

```
[ ]: ## Set to correct float type for consistency with training
tf.keras.backend.set_floatx('float32')
```

```
[ ]: ## Import necessary variational autoencoders
## and predictors
import GANS.renewed_smiles_vae as renewed_smiles_vae
import GANS.implicit_smile as implicit_smile
import GANS.proppred_deep_conv_smiles_vae as proppred_vae
import GANS.decoderTransformerLatent as decoderTransformerLatent
import GANS.ic50vae as ic50vae
import GANS.ic50pred as ic50pred
```

```
[ ]: ## Guacamol to evaluate found smiles
from typing import List
from guacamol.distribution_matching_generator import \
    DistributionMatchingGenerator
class MockGenerator(DistributionMatchingGenerator):
    """
    Mock generator that returns pre-defined molecules,
    possibly split in several calls
    """

    def __init__(self, molecules: List[str]) -> None:
        self.molecules = molecules
        self.cursor = 0

    def generate(self, number_samples: int) -> List[str]:
```

```

        end = self.cursor + number_samples

        sampled_molecules = self.molecules[self.cursor:end]
        self.cursor = end
        return sampled_molecules
    
```

[]: *## Converter to convert SMILES to Deep SMILES*
`converter = deepsmiles.Converter(rings = True, branches = True)`

[]: *## Import Necessary Data for training*
`SELFIES = False`
`DEEP = False`
`if SELFIES:`
 `train_smiles_X = np.load('./vocab/train_selfies_X.npy',allow_pickle=True)`
 `vocab =np.load('./vocab/selfies_vocab.npy',allow_pickle=True)`
 `vocab_index = np.load('./vocab/selfies_vocab_index.npy',allow_pickle=True)`
`elif DEEP:`
 `train_smiles_X = np.load('./vocab/train_deep_smiles_X.npy',allow_pickle=True)`
 `vocab =np.load('./vocab/deep_vocab.npy',allow_pickle=True)`
 `vocab_index = np.load('./vocab/deep_vocab_index.npy',allow_pickle=True)`
`else:`
 `train_smiles_X = np.load('./vocab/train_smiles_X.npy',allow_pickle=True)`
 `vocab =np.load('./vocab/vocab.npy',allow_pickle=True)`
 `vocab_index = np.load('./vocab/vocab_index.npy',allow_pickle=True)`
`vocab = dict(vocab.ravel()[0])`
`vocab_index = dict(vocab_index.ravel()[0])`

[]: *## ZINC data*
`if SELFIES:`
 `zinc_train_smiles_X = np.load('./zinc_train_selfies_actual_X.`
 `↪npy',allow_pickle=True)`
 `zinc_vocab =np.load('./vocab/zinc_selfies_vocab.npy',allow_pickle=True)`
 `zinc_vocab_index = np.load('./vocab/zinc_selfies_vocab_index.`
 `↪npy',allow_pickle=True)`
`elif DEEP:`
 `zinc_train_smiles_X = np.load('./vocab/zinc_train_deep_smiles_X.`
 `↪npy',allow_pickle=True)`
 `zinc_vocab =np.load('./vocab/zinc_deep_vocab.npy',allow_pickle=True)`
 `zinc_vocab_index = np.load('./vocab/zinc_deep_vocab_index.`
 `↪npy',allow_pickle=True)`
`else:`
 `zinc_train_smiles_X = np.load('./vocab/zinc_train_smiles_X.`
 `↪npy',allow_pickle=True)`
 `zinc_vocab =np.load('./vocab/zinc_vocab.npy',allow_pickle=True)`
 `zinc_vocab_index = np.load('./vocab/zinc_vocab_index.npy',allow_pickle=True)`
`zinc_vocab = dict(zinc_vocab.ravel()[0])`

```

zinc_vocab_index = dict(zinc_vocab_index.ravel()[0])

[ ]: import re
## replace Br and Cl with single letters
def replace_halogens_inv(string):
    br = re.compile('R')
    cl = re.compile('L')
    string = br.sub('Br', string)
    string = cl.sub('Cl', string)
    return string

[ ]: ## Splits the selfies <molecule> into a list of character strings.
def split_selfie(molecule):
    return re.findall(r'\[^.\]|\. ', molecule)

[ ]: ## Integer encode SMILES and DeepSMILES
def integer_encode(smiles,vocab_dict):
    smiles_enc = []
    for char in smiles:
        if char != '\n':
            smiles_enc.append(vocab_dict[char])
    return smiles_enc

[ ]: ## Integer encode SELFIES
def integer_encode_selfies(selfies,vocab_dict):
    selfies_enc = []
    try:
        for char in selfies:
            if char != '\n':
                selfies_enc.append(vocab_dict[char])
    except:
        return None
    return selfies_enc

[ ]: ## Get SMILES from vocabulary tokens
def get_smiles_from_tokens(tokens,vocab_index):
    text_generated = []
    for i in tokens:
        text_generated.append(vocab_index[i])
    eos_index = text_generated.index('<EOS>')
    text_generated = text_generated[1:eos_index]
    return ''.join(text_generated)

[ ]: ## Takes processed smiles/deep smiles and returns the tokenized
## versions of the smiles or deep semiles
## Note: Run replace halogens and replace percentages
## before running this method

```

```

def tokenize_smiles(smiles):
    char_list = list(smiles)
    tokenized= []
    tokenized.append('<BOS>')
    i = 0
    while i < len(char_list):
        char = char_list[i]
        tokenized.append(char)
        i= i+1
    tokenized.append('<EOS>')
    return tokenized

```

```

[ ]: import re
## replace Br and Cl with single letters
def replace_halogens(string):
    br = re.compile('Br')
    cl = re.compile('Cl')
    string = br.sub('R', string)
    string = cl.sub('L', string)
    return string

```

```

[ ]: ## Takes processed selfies smiles and returns the tokenized
## versions of the selfies
def tokenize_selfies(selfies):
    char_list = split_selfie(selfies)
    tokenized= []
    tokenized.append('<BOS>')
    i = 0
    while i < len(char_list):
        char = char_list[i]
        tokenized.append(char)
        i = i+1
    tokenized.append('<EOS>')
    return tokenized

```

```

[ ]: ## Tokenize Zinc data set
smile_pair_tokens = []
if not SELFIES:
    index = np.where(zinc_train_smiles_X == 1)
    t = np.split(zinc_train_smiles_X,index[0].tolist())
    t= t[1:]
else:
    t = zinc_train_smiles_X
if not SELFIES:
    for smiles in t:
        smiles = get_smiles_from_tokens(smiles,zinc_vocab_index)
        if SELFIES:

```

```

    smile_pair_tokens.append(tokenize_selfies(smiles))
else:
    smiles = replace_halogens(smiles)
    smile_pair_tokens.append(tokenize_smiles(smiles))
smile_pair_tokens = np.array(smile_pair_tokens)
else:
    smile_pair_tokens = zinc_train_smiles_X

smiles_ordered = []
for smiles in smile_pair_tokens:
    if SELFIES:
        smiles = smiles
        smiles_ordered.append(smiles)
    else:
        smiles_ordered.append(integer_encode(smiles, vocab))
smiles_ordered = np.array(smiles_ordered)
zinc_train_smiles_X = smiles_ordered

```

```
[ ]: ## Ensure that our tokenization works and that we do not continue
## with NULL data
zinc_train_smiles_X =[v for i,v in enumerate(zinc_train_smiles_X) if v != None]
print(len(zinc_train_smiles_X))
```

```
[ ]: ## Import sas, qed, and logp data for ZINC
if SELFIES:
    zinc_sas = np.load('./vocab/selfies_zinc_sas.npy',allow_pickle=True)
    zinc_qed =np.load('./vocab/selfies_zinc_qeds.npy',allow_pickle=True)
    zinc_logp =np.load('./vocab/selfies_zinc_logp.npy',allow_pickle=True)
else:
    zinc_sas = np.load('./vocab/zinc_sas.npy',allow_pickle=True)
    zinc_qed =np.load('./vocab/zinc_qeds.npy',allow_pickle=True)
    zinc_logp =np.load('./vocab/zinc_logp.npy',allow_pickle=True)
```

```
[ ]: ## Import sas, qed, and logp data for ChEMBL
sas = np.load('./vocab/sas.npy',allow_pickle=True)
qed =np.load('./vocab/qed.npy',allow_pickle=True)
logp =np.load('./vocab/logp.npy',allow_pickle=True)
```

```
[ ]: ## Import IC50 and transcriptomic data
gene_expressions = np.load('gene_expressions.npy')
ic50 = np.load('ic50.npy')
smiles_pairs = np.load('smiles_pairs.npy')
sites = np.load('sites.npy')
cell_lines = np.load('cell_lines.npy')
histologies = np.load('histologies.npy')
```

```
[ ]: ## Scale data as necessary
from sklearn.preprocessing import MinMaxScaler
ic50_scaler = MinMaxScaler()
ic50_scaler.fit(ic50.reshape(-1,1))
ic50 = ic50_scaler.transform(ic50.reshape(-1,1))
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(gene_expressions)
gene_expressions = scaler.transform(gene_expressions)
```

```
[ ]: ## Get IC50 data for the 'TE-12' cell line
sarcoma_gene_expressions = []
sarcoma_smiles = []
sarcoma_ic50s = []
indices = []
index = 0
for i in histologies:
    if cell_lines[index] =='TE-12':
        indices.append(index)
    index+=1
real_sampled_smiles = np.array(smiles_pairs)[indices]
sampled_smiles =  np.array(smiles_pairs)[indices]
sampled_ic50 = ic50[indices]
sampled_gene_expressions = gene_expressions[indices]

indices = []
index = 0
for i in histologies:
    if cell_lines[index] =='TE-12':
        indices.append(index)
    index+=1
real_sampled_smiles = np.array(smiles_pairs)[indices]
sampled_smiles =  np.array(smiles_pairs)[indices]
sampled_ic50 = ic50[indices]
sampled_gene_expressions = gene_expressions[indices]

smiles_pairs_enc = []
for smiles in sampled_smiles:
    if SELFIES:
        smiles_pairs_enc.append(encoder(smiles))
if SELFIES:
    sampled_smiles = np.array(smiles_pairs_enc)
print(len(sampled_smiles))
```

```
[ ]: ## Get IC50 data for the sarcoma cells
indices = []
index = 0
for i in histologies:
    if i == 'sarcoma':
        indices.append(index)
    index+=1
sarcoma_cell_lines = cell_lines[indices]
sarcoma_smiles = np.array(smiles_pairs)[indices]
sarcoma_ic50s = ic50[indices]
sarcoma_gene_expressions = gene_expressions[indices]
```

```
[ ]: ## Get sarcoma gene expression data
new_sarcoma_gene_expressions = []
new_cell_lines = []
for i in range(len(sarcoma_cell_lines)):
    if cell_lines[i] not in new_cell_lines:
        new_cell_lines.append(cell_lines[i])
        new_sarcoma_gene_expressions.append(sarcoma_gene_expressions[i])
print(len(new_sarcoma_gene_expressions))
sarcoma_gene_expressions = new_sarcoma_gene_expressions
```

```
[ ]: ## Necessary CONSTANTS
BATCH_SIZE = 256
VOCAB_SIZE = len(vocab_index)
EPOCHS = 10
LEARNING_RATE = 1e-4
if SELFIES or DEEP:
    PAD_SIZE = 250
else:
    PAD_SIZE = 160 ## Maximum size of a SMILE (100 + BOS, EOS)
    print('HERE')
MAX_LEN = PAD_SIZE
DROP_OUT= 0.2
EMBEDDING_DIM = 192 ## Embedding dim of the characters
HIDDEN_DIM = 256
DROPOUT = 0.2
TRAIN = False
LATENT_DIM = 64
IMPLICIT = False
PROPPRED = False
TRANSFORMER_DECODE = False
IC50 = True
SMALL_LATENT = False
if SMALL_LATENT:
    LATENT_DIM = 32
print(MAX_LEN)
```

```
[ ]: ## Process ZINC data
t=zinc_train_smiles_X
t = tf.keras.preprocessing.sequence.pad_sequences(t,maxlen = ↵
    ↵PAD_SIZE,padding='post')
NUM_BATCHES = math.floor(len(t)/BATCH_SIZE )

NUM_TRAIN_BATCH = math.floor(NUM_BATCHES*0.99)
NUM_TEST_BATCH = math.floor(NUM_BATCHES*(0.01))

zinc_test_X = t
zinc_test_sas = zinc_sas
zinc_test_qed = zinc_qed
zinc_test_logp = zinc_logp
```

```
[ ]: ## Process ChEMBL data
index = np.where(train_smiles_X == 1)
t = np.split(train_smiles_X,index[0].tolist())
t= t[1:]
t = tf.keras.preprocessing.sequence.pad_sequences(t,maxlen = ↵
    ↵PAD_SIZE,padding='post')
NUM_BATCHES = math.floor(len(t)/BATCH_SIZE )
```

```
[ ]: NUM_TRAIN_BATCH = math.floor(NUM_BATCHES*0.99)
NUM_TEST_BATCH = math.floor(NUM_BATCHES*(0.01))
```

```
[ ]: test_X = t[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↵(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_X = t[:NUM_TRAIN_BATCH*BATCH_SIZE]
```

```
[ ]: test_sas = sas[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↵(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_sas = sas[:NUM_TRAIN_BATCH*BATCH_SIZE]

test_qed = qed[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↵(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_qed = qed[:NUM_TRAIN_BATCH*BATCH_SIZE]

test_logp = logp[NUM_TRAIN_BATCH*BATCH_SIZE:
    ↵(NUM_TEST_BATCH+NUM_TRAIN_BATCH)*BATCH_SIZE]
train_logp = logp[:NUM_TRAIN_BATCH*BATCH_SIZE]
```

```
[ ]: ## Create transformer
transformer = decoderTransformerLatent.Transformer(batch_size = BATCH_SIZE, ↵
    ↵embedding_dim=384,embedding_dropout =DROP_OUT,max_len=MAX_LEN,
                                                num_heads = 6, num_layers = ↵
    ↵6,
```

```

        vocab_size = VOCAB_SIZE,attention_dropout_=_
→=DROP_OUT,d_hid=EMBEDDING_DIM *4,use_one_embedding_dropout=False)

[ ]: ## Load in necessary Transformer weights
if IMPLICIT and IC50:
    print('NOT TRAINED YET')
elif IMPLICIT:
    if DEEP:
        transformer.load_weights('ivae_decoding_deep_latent')
    elif SELFIES:
        transformer.load_weights('ivae_decoding_selfies_latent')
    else:
        transformer.load_weights('ivae_decoding_smiles_latent')
        print('HERE')
elif IC50:
    if DEEP:
        transformer.load_weights('ic50g_decoding_deep_latent3')
    elif SELFIES:
        transformer.load_weights('ic50g_decoding_selfies_latent3')
    else:
        transformer.load_weights('amazing_ic50g_decoding_smiles_latent3')
        print('IC50G NORMAL')
else:
    if DEEP:
        transformer.load_weights('decoding_deep_smiles_latent3')
    elif SELFIES:
        transformer.load_weights('decoding_selfies_latent3')
    else:
        transformer.load_weights('decoding_smiles_latent3')
        print('NORMAL')

```

```

[ ]: # Create necessary VAE
if IMPLICIT:
    smile_vae = implicit_smile.SMILE_IMPLICIT_VAE(vocab_size=_
→=VOCAB_SIZE,embedding_dim =EMBEDDING_DIM,
                                                max_len =MAX_LEN, latent_dim=LATENT_DIM, hidden_dim= HIDDEN_DIM,
                                                recurrent_dropout =0.2,
                                                dropout_rate=0.2,
                                                epsilon_std = 1.0)
    print('HERE')
else:
    smile_vae = renewed_smiles_vae.SMILE_VAE(vocab_size=_
→=VOCAB_SIZE,embedding_dim=EMBEDDING_DIM, max_len= MAX_LEN,
                                              latent_dim = LATENT_DIM, recurrent_dropout =_
→DROP_OUT,dropout_rate= DROP_OUT)

```

```
[ ]: ## Load in necessary VAE weights
if IMPLICIT:
    if DEEP:
        print('IMPLICIT DEEP')
        smile_vae.load_weights('deep_smiles_ivae_weights')
    elif SELFIES:
        print('IMPLICIT SELFIES')
        smile_vae.load_weights('selfies_ivae_weights')
    else:
        print('IMPLICIT NORMAL')
        smile_vae.load_weights('smiles_ivae_weights')
elif not IC50:
    if SMALL_LATENT:
        if DEEP:
            print('SMALL DEEP')
            smile_vae.load_weights('deep32_conv_vae_weights2')
        elif SELFIES:
            print('SMALL SELFIES')
            smile_vae.load_weights('selfies32_conv_vae_weights2')
        else:
            print('SMALL SMILES')
            smile_vae.load_weights('smiles32_conv_vae_weights2')
    else:
        if DEEP:
            print('DEEP')
            smile_vae.load_weights('deep_conv_vae_weights2')
        elif SELFIES:
            print('SELFIES')
            smile_vae.load_weights('selfies_conv_vae_weights2')
        else:
            print('NORMAL')
            smile_vae.load_weights('smiles_conv_vae_weights2')
if IC50:
    print('HERE')
    smile_vae = ic50vae.SMILE_VAE(vocab_size=_VOCAB_SIZE, embedding_dim=EMBEDDING_DIM, max_len= MAX_LEN,
                                    latent_dim = LATENT_DIM, recurrent_dropout =_DROP_OUT, dropout_rate= DROP_OUT)
    if DEEP:
        print('IC50 DEEP')
        smile_vae.load_weights('ic50g_deep_conv_vae_weights')
    elif SELFIES:
        print('IC50 SELFIES')
        smile_vae.load_weights('ic50g_selfies_conv_vae_weights')
    else:
        print('IC50 NORMAL')
        smile_vae.load_weights('amazing_ic50g_smiles_conv_vae_weights')
```

```
[ ]: ## converts smiles to deep smiles
def get_smiles_from_selfies(deep_list):
    smiles = []
    for deep in deep_list:
        try:
            smile = decoder(deep)
        except:
            smile = None
        smiles.append(smile)
    return smiles
```

```
[ ]: ## converts smiles to deep smiles
def get_smiles_from_deep(deep_list):
    smiles = []
    print("DeepSMILES version: %s" % deepsmiles.__version__)
    converter = deepsmiles.Converter(rings = True, branches = True)
    print(converter) # record the options used
    i = 0
    for deep in deep_list:
        print(deep)
        try:
            smile = converter.decode(deep)
        except:
            smile = None
            #print("DecodeError! Error message was '%s'" % e.message)
            continue
        if (i%100000 == 0 ):
            print(i)
        i+= 1
        smiles.append(smile)
    return smiles
```

```
[ ]: def get_smiles_from_logits_prob(logits,vocab_index, temperature =1.0):
    logits = logits/temperature
    text_generated = []
    for i in logits[0]:
        j = tf.random.categorical(tf.reshape(i,[1,len(i)]), num_samples=1)[0][0].
        ↴numpy()
        text_generated.append(vocab_index[j])
    eos_index = text_generated.index('<EOS>')
    text_generated = text_generated[1:eos_index]
    return (''.join(text_generated))
    #print(''.join(text_generated))
```

```
[ ]: ## return smile string from logits
def get_smiles_from_logits(logits,vocab_index):
    soft = tf.nn.softmax(logits, axis = -1)
```

```

prediction = tf.argmax(soft, -1)
text_generated = []
for i in prediction[0]:
    text_generated.append(vocab_index[i.numpy()])
if '<EOS>' in text_generated:
    eos_index = text_generated.index('<EOS>')
    text_generated = text_generated[1:eos_index]
else:
    text_generated = []
return (''.join(text_generated))
#print(''.join(text_generated))

```

```

[ ]: ## return smile string from logits
def get_ismiles_from_logits(logits,vocab_index):
    soft = tf.nn.softmax(logits, axis = -1)
    prediction = tf.argmax(soft, -1)
    text_generated = []
    for i in prediction[0]:
        text_generated.append(vocab_index[i.numpy()])
    eos_index = text_generated.index('<EOS>')
    text_generated = text_generated[1:eos_index]
    return (''.join(text_generated))
#print(''.join(text_generated))

```

```

[ ]: ## return smile string from logits
def get_smiles_from_logits_topp(logits,vocab_index,k =5):
    text_generated = []
    for log in logits[0]:
        ## sort predictions
        log = np.array(log)
        ind = np.array(log).argsort() [::-1] [:len(log)] ## Sort in descending
        ↵order
        sorted_prob = tf.nn.softmax(logits[ind])
        ## Get all the indicies that are less than the total allowed probability
        cumprob = np.cumsum(sorted_prob)
        ind_stay = cumprob < p
        if (len(ind_stay) <= 1):
            predicted_id = ind[0]
        else:
            sorted_logits = logits[ind]
            # using a categorical distribution to predict the character returned by
            ↵the model
            predicted_index = tf.random.categorical(tf.
            ↵reshape(sorted_logits,[1,len(sorted_logits)]), num_samples=1).numpy()
            predicted_id = ind[predicted_index][0][0]
            # using a categorical distribution to predict the character returned by the
            ↵model

```

```

#predicted_id = tf.random.categorical(tf.
→reshape(predictions,[1,len(predictions)]), num_samples=1)[0][0].numpy()
text_generated.append(vocab_index[ind[predicted_id]])
eos_index = text_generated.index('<EOS>')
text_generated = text_generated[:eos_index]
return (''.join(text_generated))
#print(''.join(text_generated))

```

```

[ ]: ## return smile string from logits
def get_smiles_from_logits_topk(logits,vocab_index,k =5):
    #soft = tf.nn.softmax(logits, axis = -1)
    #prediction = tf.argmax(soft, -1)
    text_generated = []
    for log in logits[0]:
        ## sort predictions
        log = np.array(log)
        ind = np.array(log).argsort() [::-1] [:len(log))] ## Sort in descending
→order
        ## Get sorted predictions and the top k predictions
        log = log[ind]
        log = log[:k]
        # using a categorical distribution to predict the character returned by the
→model
        predicted_id = tf.random.categorical(tf.
→reshape(predictions,[1,len(predictions)]), num_samples=1)[0][0].numpy()
        text_generated.append(vocab_index[ind[predicted_id]])
        eos_index = text_generated.index('<EOS>')
        text_generated = text_generated[:eos_index]
    return (''.join(text_generated))
#print(''.join(text_generated))

```

```

[ ]: def get_smiles_from_tokens(tokens,vocab_index):
    text_generated = []
    for i in tokens:
        text_generated.append(vocab_index[i])
    eos_index = text_generated.index('<EOS>')
    text_generated = text_generated[1:eos_index]
    return ''.join(text_generated)

```

```

[ ]: import re
## replace R and L with Br and Cl
def replace_halogens_inv(string):
    br = re.compile('R')
    cl = re.compile('L')
    string = br.sub('Br', string)
    string = cl.sub('Cl', string)
    return string

```

```
[ ]: ## Tokenize and process ChEMBL data
sarcoma_smiles = set(sarcoma_smiles)
smile_pair_tokens = []
for smiles in sarcoma_smiles:
    if SELFIES:
        smile_pair_tokens.append(tokenize_selfies(encoder(smiles)))
    elif DEEP:
        smiles = converter.encode(smiles)
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(smiles))
    else:
        smiles = replace_halogens(smiles)
        smile_pair_tokens.append(tokenize_smiles(smiles))
smile_pair_tokens = np.array(smile_pair_tokens)

smiles_ordered = []
for smiles in smile_pair_tokens:
    if SELFIES:
        smiles_ordered.append(integer_encode_selfies(smiles,vocab))
    else:
        smiles_ordered.append(integer_encode(smiles,vocab))
smiles_ordered = np.array(smiles_ordered)

smiles_ordered = tf.keras.preprocessing.sequence.
    ↪pad_sequences(smiles_ordered,maxlen = PAD_SIZE,padding='post')
```

```
[ ]: ## Process Sarcoma data
sarcoma_tok_smiles = np.array(smiles_ordered)
sarcoma_smiles_z = []
for smiles in sarcoma_tok_smiles:
    smiles = smiles.reshape(1,smiles.shape[0])[:, :]
    if IMPLICIT:
        eps = tf.convert_to_tensor(np.random.normal(size=(smiles.shape[0], ↪
        ↪LATENT_DIM)),dtype = tf.float32)
        z_mean = smile_vae.encoder(smiles,eps)[1]
    else:
        h, z_mean,z_log_var = smile_vae.encoder(np.array(smiles[:]) .
        ↪reshape(1,MAX_LEN,)) 
    sarcoma_smiles_z.append(z_mean)
```

```
[ ]: ## Process TE-12 cell line data
smile_pair_tokens = []
for smiles in sampled_smiles:
    if SELFIES:
        smile_pair_tokens.append(tokenize_selfies(smiles))
    elif DEEP:
        smiles = converter.encode(smiles)
```

```

    smiles = replace_halogens(smiles)
    smile_pair_tokens.append(tokenize_smiles(smiles))
else:
    smiles = replace_halogens(smiles)
    smile_pair_tokens.append(tokenize_smiles(smiles))
smile_pair_tokens = np.array(smile_pair_tokens)

smiles_ordered = []
for smiles in smile_pair_tokens:
    if SELFIES:
        smiles_ordered.append(integer_encode_selfies(smiles,vocab))
    else:
        smiles_ordered.append(integer_encode(smiles,vocab))
smiles_ordered = np.array(smiles_ordered)

smiles_ordered = tf.keras.preprocessing.sequence.
    pad_sequences(smiles_ordered,maxlen = PAD_SIZE,padding='post')

```

```

[ ]: ## Get Latnet representations of select TE-12 data
sampled_smiles =[]
for smiles in smiles_ordered:
    smiles = smiles.reshape(1,smiles.shape[0])[:, :]
    if IMPLICIT:
        eps = tf.convert_to_tensor(np.random.normal(size=(smiles.shape[0],LATENT_DIM)),dtype = tf.float32)
        z_mean = smile_vae.encoder(smiles,eps)[1]
    else:
        h, z_mean,z_log_var = smile_vae.encoder(np.array(smiles[:]).reshape(1,MAX_LEN,))
    sampled_smiles.append(z_mean)

sampled_smiles = np.array(sampled_smiles)
sampled_smiles = sampled_smiles.reshape((-1,LATENT_DIM))
print(sampled_smiles.shape)

```

```

[ ]: ## Process data as necessary for testing the efficacy of different models
ZINC = True
TRANSFORMER_DECODE = True
current_testing_smiles = smiles_ordered
if ZINC:
    current_testing_smiles = zinc_test_X[:1000]
else:
    current_testing_smiles = test_X[:1000]

test_smiles = []
index = 0

```

```

for test_smile in current_testing_smiles:
    if index % 10 == 0:
        print(index)
    test_s = test_smile.reshape(1,test_smile.shape[0])[::,:]
    if IMPLICIT:
        eps = tf.convert_to_tensor(np.random.normal(size=(test_s.shape[0],LATENT_DIM)),dtype = tf.float32)
        test_s_x = smile_vae.encoder(test_s,eps)[1]
    if TRANSFORMER_DECODE == True:
        zeros = np.zeros((BATCH_SIZE-1,LATENT_DIM))
        test_s_x = np.concatenate((test_s_x,zeros))
        test_s_x = transformer(test_s_x)
    else:
        test_s_x = smile_vae(test_s)[0]
        test_smiles.append(get_smiles_from_logits(test_s_x,vocab_index))
    else:
        test_s_x = smile_vae.encoder(test_s)[1]
    if TRANSFORMER_DECODE == True:
        zeros = np.zeros((BATCH_SIZE-1,LATENT_DIM))
        test_s_x = np.concatenate((test_s_x,zeros))
        test_s_x = tf.reshape(transformer(test_s_x)[0], [1,MAX_LEN,VOCAB_SIZE])
    else:
        test_s_x = smile_vae(test_s)[2]
        test_smiles.append(get_smiles_from_logits(test_s_x,vocab_index))
    index+=1
if DEEP:
    print('HERE')
    actual_smiles = get_smiles_from_deep(test_smiles)
elif SELFIES:
    actual_smiles = get_smiles_from_selfies(test_smiles)
else:
    actual_smiles = test_smiles

```

```

[ ]: ## Process back to get original SMILES
correct_smiles =[v for i,v in enumerate(actual_smiles) if v != None and v!= -1]
testing_set = [get_smiles_from_tokens(v,vocab_index) for i,v in enumerate(current_testing_smiles)]
if DEEP:
    testing_set = get_smiles_from_deep(testing_set)
elif SELFIES:
    testing_set = get_smiles_from_selfies(testing_set)

if not SELFIES:
    for i in range(len(correct_smiles)):
        correct_smiles[i] = replace_halogens_inv(correct_smiles[i])

```

```
[ ]: ## Check the number of syntactically correct smiles
num_incorrect = 0
for i in correct_smiles:
    try:
        m = rdkit.Chem.MolFromSmiles(i,sanitize=False)
        if m is None:
            num_incorrect+=1
    except:
        continue
print(1-(num_incorrect/len(current_testing_smiles)))
```

```
[ ]: NUM_SAMPLES = len(correct_smiles)

### ASSESS NUMBER OF CORRECT SMILES
generator = MockGenerator(correct_smiles)
benchmark = ValidityBenchmark(number_samples=NUM_SAMPLES)
print("Percentage of Chemically Reasonable Generated SMILES: " + \
"{:.3f}".format(benchmark.assess_model(generator).score))

## Assess generated strings
generator = MockGenerator(correct_smiles)
print(len(testing_set))
#training_set = molecules['Smiles'].tolist()

### Assess the number of unique SMILES generated
benchmark = UniquenessBenchmark(number_samples=NUM_SAMPLES)
print("The model's uniqueness score is: " + \
"{:.3f}".format(benchmark.assess_model(generator).score))

### Assess the novelty of the SMILES generated
#generator = MockGenerator(['CCOCC', 'O(CC)CC', 'C=CC=C', 'CC'])
generator = MockGenerator(correct_smiles)

benchmark = NoveltyBenchmark(number_samples=NUM_SAMPLES, \
    training_set=testing_set)
#benchmark = NoveltyBenchmark(number_samples=NUM_SAMPLES, training_set=['CO', \
    'CC'])
print("The model's novelty score is: " + \
    "{:.3f}".format(benchmark.assess_model(generator).score))

## Assess the KL-Divergence of the SMILES generated
#benchmark = KLDivBenchmark(number_samples=NUM_SAMPLES, \
    training_set=testing_set)
generator = MockGenerator(correct_smiles)
benchmark = KLDivBenchmark(number_samples=NUM_SAMPLES, training_set=testing_set)
result = benchmark.assess_model(generator)
```

```

print("The model's KL Divergence score is: " + \
      "{:.3f}".format(result.score))

[ ]: ### Perturb the latent space
def perturb_z( z, noise_norm, constant_norm=False):
    if noise_norm > 0.0:
        noise_vec = np.random.normal(0, 1, size=z.shape)
        noise_vec = noise_vec / np.linalg.norm(noise_vec)
        if constant_norm:
            return z + (noise_norm * noise_vec)
        else:
            noise_amp = np.random.uniform(
                0, noise_norm, size=(z.shape[0], 1))
            return z + (noise_amp * noise_vec)
    else:
        return z

[ ]: ## Get many SMILES from a given SMILES and VAE
def smile_to_smiles(smile_vae,
                     smile,
                     vocab_index,
                     noise_norm =0.1,
                     decode_attempts=250):
    h, z_mean,z_log_var  = smile_vae.encoder(tf.reshape(smile,[1,MAX_LEN]))
    zs = []
    for i in range(decode_attempts):
        zs.append(perturb_z(z_mean,2.0))
    zs = np.array(zs)
    zs = zs.reshape((zs.shape[0],zs.shape[-1]))
    #print(zs.shape)
    x_decoded = []
    #for z in zs:
    x_decoded = smile_vae.decoder(zs)
    decoded_smiles = []
    for x_dec in x_decoded:
        decoded_smiles.append(get_smiles_from_logits(tf.
        ↪reshape(x_dec,[1,MAX_LEN,VOCAB_SIZE]),vocab_index))
    actual_smiles = decoded_smiles
    if DEEP == True:
        actual_smiles = get_smiles_from_deep(decoded_smiles)
        correct_smiles =[v for i,v in enumerate(actual_smiles) if v != None]
        correct_smiles =[replace_halogens_inv(v) for i,v in ↪
        ↪enumerate(actual_smiles) if v != None]
    elif SELFIES:
        actual_smiles = get_smiles_from_selfies(decoded_smiles)
        correct_smiles =[v for i,v in enumerate(actual_smiles) if v != None]
    else:

```

```

    correct_smiles =[replace_halogens_inv(v) for i,v in
                     enumerate(actual_smiles) if v != None]
    num_samples = len(correct_smiles)
    generator = MockGenerator(correct_smiles)
    benchmark = ValidityBenchmark(number_samples=num_samples)
    print("Percentage of correctly Generated SMILES: " + \
          "{:.3f}".format(benchmark.assess_model(generator).score))

    generator = MockGenerator(correct_smiles)
    benchmark = UniquenessBenchmark(number_samples=num_samples)
    print("The model's uniqueness score is: " + \
          "{:.3f}".format(benchmark.assess_model(generator).score))

    return correct_smiles

```

```

[ ]: decoded_test_samples = []
test_samples = test_X[:4000]
for smiles in test_samples:
    if SELFIES:
        smiles = get_smiles_from_tokens(smiles,vocab_index)
        smiles = decoder(smiles)
    elif DEEP:
        smiles = get_smiles_from_tokens(smiles,vocab_index)
        smiles = replace_halogens_inv(smiles)
        smiles = converter.decode(smiles)
    else:
        smiles = get_smiles_from_tokens(smiles,vocab_index)
        smiles = replace_halogens_inv(smiles)
    decoded_test_samples.append(smiles)

```

```

[ ]: ## Get fingerpreints of decoded test samples
ms = [rdkit.Chem.MolFromSmiles(x) for x in decoded_test_samples]
indexes = [i for i,v in enumerate(ms) if v != None]
ms =[v for i,v in enumerate(ms) if v != None]
fps = [rdkit.Chem.RDKFingerprint(x) for x in ms]

```

```

[ ]: ## Project test samples to latent space and perform
      ## PCA to get the distribution of Tanimoto similarities
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
from sklearn.decomposition import KernelPCA
NUM_SAMPLED = len(fps)
sampled_points = []
for i in indexes:

```

```

if IMPLICIT:
    t = smile_vae(test_X[i] [:].reshape(1,MAX_LEN))[2]
else:
    t = smile_vae(test_X[i] [:].reshape(1,MAX_LEN))[1]
sampled_points.append(t)
sampled_points = np.array(sampled_points).reshape(NUM_SAMPLED,LATENT_DIM)
PCA = KernelPCA(n_components=2, kernel="linear")
data_transformed = PCA.fit_transform(sampled_points)
from sklearn.preprocessing import MinMaxScaler
scaler1 = MinMaxScaler()
scaler1.fit(np.array(data_transformed[:,0]).reshape(-1,1))
scaler2 = MinMaxScaler()
scaler2.fit(np.array(data_transformed[:,1]).reshape(-1,1))

plot = {0: [scaler1.transform(x[0].reshape(1,-1))[0][0] for x in
            data_transformed],
        1: [scaler2.transform(x[1].reshape(1,-1))[0][0] for x in
            data_transformed]}

```

```

[ ]: ## Select the point on which to base similarity calculations
CURRENT_INDEX = np.argmin(plot[0])
if SELFIES:
    CURRENT_SMILE = decoder(get_smiles_from_tokens(test_X[:4000][indexes[CURRENT_INDEX]],vocab_index))
elif DEEP:
    CURRENT_SMILE = converter.decode(get_smiles_from_tokens(test_X[:4000][indexes[CURRENT_INDEX]],vocab_index))
else:
    CURRENT_SMILE = get_smiles_from_tokens(test_X[:4000][indexes[CURRENT_INDEX]],vocab_index)

```

```

[ ]: ## Calculate similarities
from rdkit import DataStructs
ts = []
for idx in range(len(fps)):
    ts.append(DataStructs.FingerprintSimilarity(fps[CURRENT_INDEX],fps[idx]))

```

```

[ ]: ## Plot Tanimoto similiarites as a hexplot
def scatter_hist(x, y, colors, ax, ax_histx, ax_histy):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the hexbin plot:
    ax.hexbin(x,y,gridsize= 50,C =colors, reduce_C_function=np.mean)

    ax.annotate(CURRENT_SMILE,

```

```

        (plot[0][CURRENT_INDEX]+.05, plot[1][CURRENT_INDEX]-
        ↪.05),color='red',fontsize=12 )

# now determine nice limits by hand:
#binwidth = 0.000025
#xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
#lim = (int(xymax/binwidth) + 1) * binwidth

bins = np.arange(0, 1, 0.01)
ax_histx.hist(x, bins=bins)
bins = np.arange(0, 1, 0.01)
ax_histy.hist(y, bins=bins, orientation='horizontal')

```

```

[ ]: # definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
spacing = 0.005
import matplotlib
matplotlib.rcParams['font', size=16)
ticks=np.arange(0,1.01,.1)

rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom + height + spacing, width, 0.2]
rect_histy = [left + width + spacing, bottom, 0.2, height]
print(rect_histy)
# start with a square Figure
fig = plt.figure(figsize=(12,12))
ax = fig.add_axes(rect_scatter)
cbaxes = fig.add_axes([1.00, 0.1, 0.03, 0.8])

ax_histx = fig.add_axes(rect_histx, sharex=ax)
ax_histy = fig.add_axes(rect_histy, sharey=ax)

scatter_hist(plot[0], plot[1],ts, ax, ax_histx, ax_histy)
#plt.xlim(-3, 3)
#plt.ylim(-1, 10)
norm = matplotlib.colors.Normalize(vmin=0,vmax=1 )
bar = fig.colorbar(cm.ScalarMappable(norm=norm), ax=ax,cax =cbaxes,)
bar.set_label('Tanimoto Similarity')
bar.ax.set_yticklabels(['0.0','0.2','0.4','0.6','0.8','1.0'])
#plt.axis('square')

```

```

[ ]: fig.savefig("./FINAL FIGURES/Tanimoto_IMPLICIT_DEEP_CHEMBL_TEST_4000",  

    ↪bbox_inches='tight')

[ ]: ## Project test samples to latent space and perform  

    ## PCA to get the distribution of LOGP, QED, and SA Scores for ChEMBL  

    from sklearn.decomposition import KernelPCA  

NUM_SAMPLED = 4000  

sampled_index = np.random.choice(len(test_X), NUM_SAMPLED+100 , replace=False)  

sampled_points = []  

qeds = []  

sass = []  

logps = []  

index = 0  

while len(sampled_points) != NUM_SAMPLED:  

    i = sampled_index[index]  

    index+=1  

    smiles = test_X[i][:].reshape(1,MAX_LEN)  

    eps = tf.convert_to_tensor(np.random.normal(size=(smiles.shape[0],  

    ↪LATENT_DIM)),dtype = tf.float32)  

    if IMPLICIT:  

        t = smile_vae.encoder(smiles,eps)[1]  

    else:  

        t = smile_vae.encoder(smiles)[1]  

    if SELFIES:  

        smiles = get_smiles_from_tokens(smiles[0],vocab_index)  

        try:  

            smiles = decoder(smiles)  

        except:  

            print('EXCEPT')  

            continue  

    elif DEEP:  

        smiles = get_smiles_from_tokens(smiles[0],vocab_index)  

        smiles = replace_halogens_inv(smiles)  

        smiles = converter.decode(smiles)  

    else:  

        smiles = get_smiles_from_tokens(smiles[0],vocab_index)  

        smiles = replace_halogens_inv(smiles)  

    if len(sampled_points)% 100 == 0:  

        print(len(sampled_points))  

    sampled_points.append(t)  

    m = rdkit.Chem.MolFromSmiles(smiles)  

    sass.append(sascorer.calculateScore(m))  

    qeds.append(rdkit.Chem.QED.qed(m))  

    logps.append(Descriptors.MolLogP(m))

sampled_points = np.array(sampled_points).reshape(NUM_SAMPLED,LATENT_DIM)
PCA = KernelPCA(n_components=2, kernel="linear")

```

```

data_transformed = PCA.fit_transform(sampled_points)
from sklearn.preprocessing import MinMaxScaler
scaler1 = MinMaxScaler()
scaler1.fit(np.array(data_transformed[:,0]).reshape(-1,1))
scaler2 = MinMaxScaler()
scaler2.fit(np.array(data_transformed[:,1]).reshape(-1,1))

```

```

[ ]: def scatter_hist(x, y, colors, ax, ax_histx, ax_histy):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the hexbin plot:
    ax.hexbin(x,y,gridsize=50,C =colors, reduce_C_function=np.mean)

    bins = np.arange(0, 1, 0.01)
    ax_histx.hist(x, bins=bins)
    bins = np.arange(0, 1, 0.01)
    ax_histy.hist(y, bins=bins, orientation='horizontal')

```

```

[ ]: SASS = False
logP = True

```

```

[ ]: plot = {0: [scaler1.transform(x[0].reshape(1,-1))[0][0] for x in
    ↪data_transformed],
    1: [scaler2.transform(x[1].reshape(1,-1))[0][0] for x in
    ↪data_transformed]}

```

```

[ ]: # definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
spacing = 0.005
import matplotlib
if SASS:
    ticks=np.arange(1,10,1)
    norm = matplotlib.colors.Normalize(vmin=1,vmax=10 )
elif logP:
    ticks=np.arange(-9,13+1,2)
    norm = matplotlib.colors.Normalize(vmin=-9,vmax=13 )
else:
    ticks=np.arange(0,1.2,.2)
    norm = matplotlib.colors.Normalize(vmin=0,vmax= 1)
matplotlib.rc('font', size=20)

```

```

rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom + height + spacing, width, 0.2]
rect_histy = [left + width + spacing, bottom, 0.2, height]
print(rect_histy)
# start with a square Figure
fig = plt.figure(figsize=(12,12))
ax = fig.add_axes(rect_scatter)
cbaxes = fig.add_axes([1.00, 0.1, 0.03, 0.8])

ax_histx = fig.add_axes(rect_histx, sharex=ax)
ax_histy = fig.add_axes(rect_histy, sharey=ax)
if SASS:
    scatter_hist(plot[0], plot[1],sass, ax, ax_histx, ax_histy)
elif logP:
    scatter_hist(plot[0], plot[1],logps, ax, ax_histx, ax_histy)
else:
    scatter_hist(plot[0], plot[1],qeds, ax, ax_histx, ax_histy)
#plt.xlim(-3, 3)
#plt.ylim(-1, 10)
bar =fig.colorbar(cm.ScalarMappable(norm=norm), ax=ax,cax =cbaxes,ticks=ticks)
#bar.ax.set_yticks()

bar.ax.set_yticklabels(ticks)
if SASS:
    bar.set_label('SA Scores')
elif logP:
    bar.set_label('logP')
else:
    bar.set_label('QED')
    bar.ax.set_yticklabels(['0.0','0.2','0.4','0.6','0.8','1.0'])

#plt.axis('square')
plt.show()

```

```

[ ]: fig.savefig("./FINAL FIGURES/LOGP_DEEP_CHEMBL_TEST_4000", bbox_inches='tight')

[ ]: ## Project test samples to latent space and perform
## PCA to get the distribution of LOGP, QED, and SA Scores for ZINC
from sklearn.decomposition import KernelPCA
NUM_SAMPLED = 4000
sampled_index = np.random.choice(len(zinc_test_X), NUM_SAMPLED+100 ,random_state=42,  

replace=False)
sampled_points = []
qed = []
sass = []

```

```

logps = []
index = 0
while len(sampled_points) != NUM_SAMPLED:
    i = sampled_index[index]
    index+=1
    smiles = zinc_test_X[i][:].reshape(1,MAX_LEN)
    eps = tf.convert_to_tensor(np.random.normal(size=(smiles.shape[0],LATENT_DIM)),dtype = tf.float32)
    if IMPLICIT:
        t = smile_vae.encoder(smiles,eps)[1]
    else:
        t = smile_vae.encoder(smiles)[1]
    if SELFIES:
        smiles = get_smiles_from_tokens(smiles[0],vocab_index)
        try:
            smiles = decoder(smiles)
        except:
            continue
    elif DEEP:
        smiles = get_smiles_from_tokens(smiles[0],vocab_index)
        smiles = replace_halogens_inv(smiles)
        smiles = converter.decode(smiles)
    else:
        smiles = get_smiles_from_tokens(smiles[0],vocab_index)
        smiles = replace_halogens_inv(smiles)
    if len(sampled_points)% 100 == 0:
        print(len(sampled_points))
    sampled_points.append(t)
    m = rdkit.Chem.MolFromSmiles(smiles)
    sass.append(sascorer.calculateScore(m))
    qeds.append(rdkit.Chem.QED.qed(m))
    logps.append(Descriptors.MolLogP(m))
sampled_points = np.array(sampled_points).reshape(NUM_SAMPLED,LATENT_DIM)
PCA = KernelPCA(n_components=2, kernel="linear")
data_transformed = PCA.fit_transform(sampled_points)
from sklearn.preprocessing import MinMaxScaler
scaler1 = MinMaxScaler()
scaler1.fit(np.array(data_transformed[:,0]).reshape(-1,1))
scaler2 = MinMaxScaler()
scaler2.fit(np.array(data_transformed[:,1]).reshape(-1,1))

```

```

[ ]: plot = {0: [scaler1.transform(x[0].reshape(1,-1))[0][0] for x in data_transformed],
           1: [scaler2.transform(x[1].reshape(1,-1))[0][0] for x in data_transformed]}

```

```
[ ]: def scatter_hist(x, y, colors, ax, ax_histx, ax_histy):
    # no labels
    ax_histx.tick_params(axis="x", labelbottom=False)
    ax_histy.tick_params(axis="y", labelleft=False)

    # the hexbin plot:
    ax.hexbin(x,y,gridsize=50,C =colors, reduce_C_function=np.mean)

    # now determine nice limits by hand:
    #binwidth = 0.000025
    #xymax = max(np.max(np.abs(x)), np.max(np.abs(y)))
    #lim = (int(xymax/binwidth) + 1) * binwidth

    bins = np.arange(0, 1, 0.01)
    ax_histx.hist(x, bins=bins)
    bins = np.arange(0, 1, 0.01)
    ax_histy.hist(y, bins=bins, orientation='horizontal')
```

```
[ ]: SASS = False
logP = True
```

```
[ ]: # definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
spacing = 0.005
import matplotlib
if SASS:
    ticks=np.arange(1,10,1)
    norm = matplotlib.colors.Normalize(vmin=1,vmax=10 )
elif logP:
    ticks=np.arange(-9,13+1,2)
    norm = matplotlib.colors.Normalize(vmin=-9,vmax=13 )
else:
    ticks=np.arange(0,1.2,.2)
    norm = matplotlib.colors.Normalize(vmin=0,vmax= 1)
matplotlib.rc('font', size=16)

rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom + height + spacing, width, 0.2]
rect_histy = [left + width + spacing, bottom, 0.2, height]
print(rect_histy)
# start with a square Figure
fig = plt.figure(figsize=(12,12))
```

```

ax = fig.add_axes(rect_scatter)
cbaxes = fig.add_axes([1.00, 0.1, 0.03, 0.8])

ax_histx = fig.add_axes(rect_histx, sharex=ax)
ax_histy = fig.add_axes(rect_histy, sharey=ax)
if SASS:
    scatter_hist(plot[0], plot[1], sass, ax, ax_histx, ax_histy)
elif logP:
    scatter_hist(plot[0], plot[1], logps, ax, ax_histx, ax_histy)
else:
    scatter_hist(plot[0], plot[1], qeds, ax, ax_histx, ax_histy)
#plt.xlim(-3, 3)
#plt.ylim(-1, 10)
bar = fig.colorbar(cm.ScalarMappable(norm=norm), ax=ax, cax=cbaxes,
                   ticks=ticks) #bar.ax.set_yticks()

bar.ax.set_yticklabels(ticks)
if SASS:
    bar.set_label('SA Scores')
elif logP:
    bar.set_label('logP')
else:
    bar.set_label('QED')
bar.ax.set_yticklabels(['0.0', '0.2', '0.4', '0.6', '0.8', '1.0'])

#plt.axis('square')
plt.show()

```

```
[ ]: fig.savefig("./FINAL_FIGURES/LOGP_IMPLICIT_SMILES_ZINC_4000", bbox_inches='tight')
```

```

[ ]: ## Start Optimization
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import numpy as np
from numpy.random import multivariate_normal
import GPyOpt
from GPyOpt.methods import BayesianOptimization

sampled_smiles = np.array(sampled_smiles)
sampled_smiles = sampled_smiles.reshape(len(sampled_smiles), LATENT_DIM)

```

```
[ ]: import GANS.ic50predictions as ic50predictions
from IPython.display import SVG, display
```

```

from rdkit.Chem import AllChem as Chem
from rdkit.Chem import PandasTools

[ ]: ## Load in necessary file for prediction
ic50pred = ic50predictions.IC50_MCA(vocab_size=VOCAB_SIZE,
                                      embedding_dim =EMBEDDING_DIM, num_genes =2128,
                                      hidden_dim = HIDDEN_DIM, max_len = MAX_LEN,
                                      latent_dim = LATENT_DIM)
if SELFIES:
    ic50pred.load_weights('new_ic50network_selfies')
else:
    ic50pred.load_weights('new_updated_ic50g_ic50network_smiles_basic')

[ ]: ## Create domain bounds ##
bounds = [(-4,4)]*LATENT_DIM
domains = []
for idx in range(len(bounds)):
    domain = {'name': 'latent'+str(idx), 'type': 'continuous'}
    domain['domain'] = bounds[idx]
    domains.append(domain)

[ ]: def obj_function(z):
    x = 0
    val = ic50pred(encoded_smiles = z, genes = sampled_gene_expressions[0].
    ↪reshape(1,2128)).numpy()[0][0] - x
    print(val)
    return val

[ ]: UBO = BayesianOptimization(f=obj_function, X =sampled_smiles,Y =sampled_predictions,
                                ↪domain=domains,batch_size=1,initial_design_numdata=0)

[ ]: MAX_ITER = 500
UBO.run_optimization(max_iter = MAX_ITER)

[ ]: UBO.plot_convergence()

[ ]: ins = np.array(UBO.X).tolist()
print(len(ins))
outs = np.array(UBO.Y ).tolist()
print(len(outs))
outs, ins = (list(t) for t in zip(*sorted(zip(outs, ins)))) 

[ ]: print("Value of z that minimises the objective:" + str(UBO.x_opt))
print("Maximum value of the objective: "+str(UBO.fx_opt))

```

```
[ ]: def smiles_check(smiles):
    try:
        mol = rdkit.Chem.MolFromSmiles(smiles)
        if mol is not None:
            return smiles
    except:
        pass
    return None

[ ]: def smiles_to_mol(smiles):
    try:
        mol = rdkit.Chem.MolFromSmiles(smiles)
        return mol
    except:
        return None

[ ]: ## Perform conversion of latent space back to original SMILES
def smile_to_smiles_percentage(z,norm,decode_attempts=256):
    z_mean = z
    zs = []
    #decode_attempts = 256
    for i in range(decode_attempts):
        zn = perturb_z(z_mean,norm)
        zs.append(zn)
    zs = np.array(zs)
    zs = zs.reshape((zs.shape[0],zs.shape[-1]))
    x_decoded = []
    if TRANSFORMER_DECODE:
        x_decoded = transformer(zs)
    else:
        x_decoded = smile_vae.decoder(zs)
    decoded_smiles = []
    for x_dec in x_decoded:
        decoded_smiles.append(get_smiles_from_logits(tf.
→reshape(x_dec,[1,MAX_LEN,VOCAB_SIZE]),vocab_index))
    actual_smiles = decoded_smiles
    if DEEP == True:
        actual_smiles = get_smiles_from_deep(decoded_smiles)
        correct_smiles =[replace_halogens_inv(v) for i,v in
→enumerate(actual_smiles) if v != None and v != '' ]
    elif SELFIES:
        actual_smiles = get_smiles_from_selfies(decoded_smiles)
        correct_smiles =[v for i,v in enumerate(actual_smiles) if v != None]
    else:
        correct_smiles =[replace_halogens_inv(v) for i,v in
→enumerate(actual_smiles) if v != None and v != '' ]
    actual_correct_smiles = []
```

```

for smiles in correct_smiles:
    actual_correct_smiles.append(smiles_check(smiles))
actual_correct_smiles =[v for i,v in enumerate(actual_correct_smiles) if v != None ]
num_samples = len(actual_correct_smiles)
generator = MockGenerator(actual_correct_smiles)
benchmark = UniquenessBenchmark(number_samples=num_samples)
#print(benchmark.assess_model(generator).score*decode_attempts)
return actual_correct_smiles

```

```

[ ]: def encode(smiles_list):
means = []
for smiles in smiles_list:
    tok_smile = None
    if SELFIES:
        tok_smile = tokenize_selfies(smiles)
    else:
        tok_smile = replace_halogens(smiles)
        tok_smile = tokenize_smiles(tok_smile)
    if SELFIES:
        smile_ordered = integer_encode_selfies(tok_smile,vocab)
    else:
        smile_ordered = integer_encode(tok_smile,vocab)
    t = tf.keras.preprocessing.sequence.pad_sequences(np.array(smile_ordered).
→reshape(1,len(smile_ordered)), maxlen = PAD_SIZE,padding='post')
    h, z_mean,z_log_var = smile_vae.encoder(np.array(t))
    means.append(z_mean)
return means

```

```

[ ]: import pandas as pd

def smiles_distance_z(smiles_list, z0):
    dists = []
    z_reps = encode(smiles_list)
    for z_rep in z_reps:
        dists.append(np.linalg.norm(z0 - z_rep, axis=None))
    return np.array(dists)

def ic50_dfcalc(smiles_list):
    vals = []
    z_reps = encode(smiles_list)
    for z_rep in z_reps:
        if z_rep == None:
            vals.append(None)
        else:
            val = ic50pred(encoded_smiles = z_rep, genes =
→sampled_gene_expressions[0].reshape(1,2128)).numpy()[0][0]

```

```

    val = ic50_scaler.inverse_transform(np.array(val).reshape(-1,1))
    vals.append(val[0])
return np.array(vals)

def sas_dfcalc(smiles_list):
    vals = []
    for smiles in smiles_list:
        try:
            m = rdkit.Chem.MolFromSmiles(smiles)
            if m is not None:
                vals.append(sascorer.calculateScore(m))
            else:
                vals.append(None)
        except:
            vals.append(None)
    return np.array(vals)

def qed_dfcalc(smiles_list):
    vals = []
    for smiles in smiles_list:
        try:
            m = rdkit.Chem.MolFromSmiles(smiles)
            if m is not None:
                vals.append(rdkit.Chem.QED.qed(m))
            else:
                vals.append(None)
        except:
            vals.append(None)
    return np.array(vals)

def logp_dfcalc(smiles_list):
    vals = []
    for smiles in smiles_list:
        try:
            m = rdkit.Chem.MolFromSmiles(smiles)
            if m is not None:
                vals.append(Descriptors.MolLogP(m))
            else:
                vals.append(None)
        except:
            vals.append(None)
    return np.array(vals)

## Method to display newly discovered SMILES
def prep_mol_df_i( smiles ):
    df = pd.DataFrame({'smiles': smiles})

```

```

sort_df = pd.DataFrame(df[['smiles']].groupby(
    by='smiles').size().rename('count').reset_index())

df = df.merge(sort_df, on='smiles')
df = df.drop_duplicates(subset='smiles')
df = df[df['smiles'] != '']
df.reset_index(drop=True, inplace=True)
#df = df[df['smiles'].apply(fast_verify)]
if len(df) > 0:
    df['mol'] = df['smiles'].apply(smiles_to_mol)
if len(df) > 0:
    df = df[pd.notnull(df['mol'])]
t = df['smiles']
if len(df) > 0:
    #df['distance'] = smiles_distance_z(t, z)
    df['frequency'] = df['count'] / float(sum(df['count']))
    df['IC50'] = ic50_dfcalc(t)
    df['QED'] = qed_dfcalc(t)
    df['LOGP'] = logp_dfcalc(t)
    df['SAS'] = sas_dfcalc(t)
    df = df[['smiles', 'count', 'frequency', 'IC50', 'QED', 'LOGP', 'SAS', 'mol']]
    df.sort_values(by='IC50', inplace=True)
    df.reset_index(drop=True, inplace=True)
return df

def prep_mol_df( smiles, z):
    df = pd.DataFrame({'smiles': smiles})
    sort_df = pd.DataFrame(df[['smiles']].groupby(
        by='smiles').size().rename('count').reset_index())

    df = df.merge(sort_df, on='smiles')
    df = df.drop_duplicates(subset='smiles')
    df = df[df['smiles'] != '']
    df.reset_index(drop=True, inplace=True)
    #df = df[df['smiles'].apply(fast_verify)]
    if len(df) > 0:
        df['mol'] = df['smiles'].apply(smiles_to_mol)
    if len(df) > 0:
        df = df[pd.notnull(df['mol'])]
    t = df['smiles']
    if len(df) > 0:
        df['distance'] = smiles_distance_z(t, z)
        df['frequency'] = df['count'] / float(sum(df['count']))
        df['IC50'] = ic50_dfcalc(t)
        df['QED'] = qed_dfcalc(t)
        df['LOGP'] = logp_dfcalc(t)
        df['SAS'] = sas_dfcalc(t)

```

```

        df = df[['smiles', 'distance', 'count',
→'frequency','IC50','QED','LOGP','SAS','mol']]  

        df.sort_values(by='IC50', inplace=True)  

        df.reset_index(drop=True, inplace=True)  

    return df

```

```

[ ]: ## Perform genetic algorithm for optimization:  

## We keep the top 500 examples:  

def get_trad_score(compound):  

    m = rdkit.Chem.MolFromSmiles(compound)  

    sas = sascorer.calculateScore(m)  

    qed = rdkit.Chem.QED.qed(m)  

    return 5*qed -sas

```

```

[ ]: def get_trad_scores(compounds):  

    scores = []  

    for compound in compounds:  

        scores.append(get_trad_score(compound))  

    return np.array(scores)

```

```

[ ]: ## Perform genetic algorithm for optimization:  

## We keep the top 500 examples:  

def get_ic50s(latents, gene_expression):  

    num_gene_expressions = []  

    for i in range(len(latents)):  

        num_gene_expressions.append(gene_expression)  

    num_gene_expressions = np.array(num_gene_expressions)  

    while True:  

        try:  

            ic50s_preds = ic50pred(encoded_smiles=np.array(latents), genes
→=num_gene_expressions.reshape(len(latents), 2128))  

        except:  

            print('IC50 PREDICTION EXCEPTION')  

            continue  

        break  

    return ic50s_preds

def get_ic50s_mult(latents, gene_expressions):  

    avg_ic50s = []  

    for j in range(len(latents)):  

        g_avg_ic50s = []  

        num_latents = []  

        for i in range(len(gene_expressions)):  

            num_latents.append(latents[j])  

        num_latents = np.array(num_latents)  

        while True:  

            try:

```

```

        g_avg_ic50s.append(ic50pred(encoded_smiles=np.array(num_latents),genes=
→=gene_expressions.reshape(len(gene_expressions),2128)))
    except:
        print('IC50 PREDICTION EXCEPTION')
        continue
    break
avg_ic50s.append(np.average(g_avg_ic50s))
return avg_ic50s

```

```

[ ]: ## Genetic algorithm for optimizing ic50
current_smiles = list(np.array(sampled_smiles).reshape(-1,LATENT_DIM))
decoded_smiles = list(np.array(real_sampled_smiles))
gene_expression = np.array(sampled_gene_expressions[66])
top_ic50s = list(get_ic50s(current_smiles, gene_expression))
norm = 10.0
num_generations = 10
num_out = 100
num_children = 10
for generation in range(num_generations):
    new_current_smiles = []
    new_decoded_smiles = []
    new_ic50s = []
    idx = 0
    if len(current_smiles) != 0:
        for i in range(num_children):
            parent1 = np.random.randint(0,len(current_smiles))
            parent2 = np.random.randint(0,len(current_smiles))
            diff = np.random.uniform(0, 1.0)
            child1 = np.array(current_smiles[parent1])*diff + (1-diff)*np.
→array(current_smiles[parent2])
            child2 = np.array(current_smiles[parent1])*(1-diff)+ diff*np.
→array(current_smiles[parent2])

            ## Add child1
            new_smiles = smile_to_smiles_percentage(np.array(child1).
→reshape(-1,LATENT_DIM),1.0)
            new_smiles = set(new_smiles)
            cnew_decoded_smiles = []
            cnew_decoded_smiles.extend(new_smiles)
            new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
            if new_smiles_enc.shape[0] != 0:
                print('New Child')
                ic50_preds = list(get_ic50s(new_smiles_enc,gene_expression))
                new_ic50s.extend(ic50_preds)
                new_current_smiles.extend(list(new_smiles_enc))
                new_decoded_smiles.extend(cnew_decoded_smiles)

```

```

## Add child2
new_smiles = smile_to_smiles_percentage(np.array(child2).
→reshape(-1,LATENT_DIM),1.0)
new_smiles = set(new_smiles)
cnew_decoded_smiles = []
cnew_decoded_smiles.extend(new_smiles)
new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
if new_smiles_enc.shape[0] != 0:
    print('New Child')
    ic50_preds = list(get_ic50s(new_smiles_enc,gene_expression))
    new_ic50s.extend(ic50_preds)
    new_current_smiles.extend(list(new_smiles_enc))
    new_decoded_smiles.extend(cnew_decoded_smiles)
if len(new_ic50s) != 0:
    print("BEST CHILD: " +new_decoded_smiles[np.argmin(new_ic50s)])
    print("BEST CHILD IC50: {:.5f}".format(min(new_ic50s)[0])+'\n')
for smiles in current_smiles:
    #norm = np.random.uniform(0, jump_size)
    new_smiles = smile_to_smiles_percentage(np.array(smiles).
→reshape(-1,LATENT_DIM),norm)
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = list([decoded_smiles[idx]])
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    new_smiles_enc = np.concatenate((np.array(smiles).
→reshape(-1,LATENT_DIM),new_smiles_enc))
    ic50_preds = list(get_ic50s(new_smiles_enc,gene_expression))
    new_ic50s.extend(ic50_preds)
    new_current_smiles.extend(list(new_smiles_enc))
    new_decoded_smiles.extend(cnew_decoded_smiles)
    print("BEST CURRENT EVOLUTION: "+new_decoded_smiles[np.argmin(new_ic50s)])
    print("BEST EVOLUTION IC50 : {:.5f}".format(min(new_ic50s)[0])+'\n')
    idx +=1
print('THIS GENERATION:')
decoded_smiles.extend(new_decoded_smiles)
current_smiles.extend(new_current_smiles)
top_ic50s.extend(new_ic50s)
top_ic50s, current_smiles, decoded_smiles = (list(t) for t in_
→zip(*sorted(zip(top_ic50s, np.array(current_smiles).tolist(), np.
→array(decoded_smiles).tolist() ))))
non_repeat_decoded_smiles = []
indexes = []
index = 0
for smiles in decoded_smiles:
    if smiles not in non_repeat_decoded_smiles:
        non_repeat_decoded_smiles.append(smiles)

```

```

        indexes.append(index)
        index+=1
decoded_smiles = np.array(decoded_smiles)[indexes].tolist()
top_ic50s = np.array(top_ic50s)[indexes].tolist()
current_smiles = np.array(current_smiles)[indexes].tolist()
top_ic50s = top_ic50s[:num_out]
current_smiles = current_smiles[:num_out]
decoded_smiles = decoded_smiles[:num_out]
print("BEST GENERATION : "+ decoded_smiles[np.argmin(top_ic50s)])
print("BEST GENERATION IC50 : {:.5f}".format(min(top_ic50s)[0]) +'\n')
print(decoded_smiles)

```

```

[ ]: ## Genetic algorithm for optimizing SAS/QED

#current_smiles = list(np.array(sampled_smiles).reshape(-1,LATENT_DIM))
current_smiles = current_smiles
decoded_smiles = decoded_smiles
#decoded_smiles = list(np.array(real_sampled_smiles))
top_trads = list(get_trad_scores(decoded_smiles))
norm = 10.0
num_generations = 1
num_out = 100
num_children = 10
for generation in range(num_generations):
    new_current_smiles = []
    new_decoded_smiles = []
    new_trads = []
    idx = 0
    if len(current_smiles) != 0:
        for i in range(num_children):
            parent1 = np.random.randint(0,len(current_smiles))
            parent2 = np.random.randint(0,len(current_smiles))
            diff = np.random.uniform(0, 1.0)
            child1 = np.array(current_smiles[parent1])*diff + (1-diff)*np.
            ↪array(current_smiles[parent2])
            child2 = np.array(current_smiles[parent1])*(1-diff)+ diff*np.
            ↪array(current_smiles[parent2])

        try:
            ## Add child1
            new_smiles = smile_to_smiles_percentage(np.array(child1).
            ↪reshape(-1,LATENT_DIM),1.0)
            new_smiles = set(new_smiles)
            cnew_decoded_smiles = []
            cnew_decoded_smiles.extend(new_smiles)
            new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
            if new_smiles_enc.shape[0] != 0:

```

```

print('New Child')
trad_scores = list(get_trad_scores(cnew_decoded_smiles))
new_trads.extend(trad_scores)
new_current_smiles.extend(list(new_smiles_enc))
new_decoded_smiles.extend(cnew_decoded_smiles)
except:
    continue

## Add child2
try:
    new_smiles = smile_to_smiles_percentage(np.array(child2).
→reshape(-1,LATENT_DIM),1.0)
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = []
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    if new_smiles_enc.shape[0] != 0:
        print('New Child')
        trad_scores = list(get_trad_scores(cnew_decoded_smiles))
        new_trads.extend(trad_scores)
        new_current_smiles.extend(list(new_smiles_enc))
        new_decoded_smiles.extend(cnew_decoded_smiles)
    except:
        continue
    if len(new_trads) != 0:
        print("BEST CHILD: " +new_decoded_smiles[np.argmax(new_trads)])
        print("BEST CHILD TRAD: {:.5f}".format(max(new_trads))+'\n')
for smiles in current_smiles:
    #norm = np.random.uniform(0, jump_size)
    new_smiles = smile_to_smiles_percentage(np.array(smiles).
→reshape(-1,LATENT_DIM),norm)
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = list([decoded_smiles[idx]])
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    new_smiles_enc = np.concatenate((np.array(smiles).
→reshape(-1,LATENT_DIM),new_smiles_enc))
    trad_scores = list(get_trad_scores(cnew_decoded_smiles))
    new_trads.extend(trad_scores)
    new_current_smiles.extend(list(new_smiles_enc))
    new_decoded_smiles.extend(cnew_decoded_smiles)
    print("BEST CURRENT EVOLUTION: "+new_decoded_smiles[np.argmax(new_trads)])
    print("BEST EVOLUTION TRAD : {:.5f}".format(max(new_trads))+'\n')
    idx +=1
print('THIS GENERATION:')
decoded_smiles.extend(new_decoded_smiles)

```

```

current_smiles.extend(new_current_smiles)
top_trads.extend(new_trads)
top_trads, current_smiles, decoded_smiles = (list(t) for t in
zip(*sorted(zip(top_trads, np.array(current_smiles).tolist(), np.
array(decoded_smiles).tolist()), reverse=True)))
non_repeat_decoded_smiles = []
indexes = []
index = 0
for smiles in decoded_smiles:
    if smiles not in non_repeat_decoded_smiles:
        indexes.append(index)
        index+=1
decoded_smiles =np.array(decoded_smiles)[indexes].tolist()
top_trads =np.array(top_trads)[indexes].tolist()
current_smiles =np.array(current_smiles)[indexes].tolist()
top_trads = top_trads[:num_out]
current_smiles = current_smiles[:num_out]
decoded_smiles = decoded_smiles[:num_out]
print("BEST GENERATION : "+ decoded_smiles[np.argmax(top_trads)])
print("BEST GENERATION TRAD : {:.5f}".format(max(top_trads)) +'\n')
print(decoded_smiles)

```

```

[ ]: top_trads = top_trads[:num_out]
current_smiles = current_smiles[:num_out]
decoded_smiles = decoded_smiles[:num_out]
print("BEST GENERATION : "+ decoded_smiles[np.argmax(top_trads)])
print("BEST GENERATION TRAD : {:.5f}".format(max(top_trads)) +'\n')
print(decoded_smiles)

```

```

[ ]: ## MULT
## Genetic algorithm for optimizing Sarcoma drugs

current_smiles = list(np.array(sarcoma_smiles_z).reshape(-1,LATENT_DIM))
decoded_smiles = list(sarcoma_smiles)
gene_expression = np.array(sarcoma_gene_expressions)
top_ic50s = list(get_ic50s_mult(current_smiles, gene_expression))
norm = 10.0
num_generations = 10
num_out = 100
num_children = 10
for generation in range(num_generations):
    new_current_smiles =[]
    new_decoded_smiles = []
    new_ic50s = []
    idx = 0
    if len(current_smiles) != 0:
        for i in range(num_children):

```

```

parent1 = np.random.randint(0,len(current_smiles))
parent2 = np.random.randint(0,len(current_smiles))
diff = np.random.uniform(0, 1.0)
child1 = np.array(current_smiles[parent1])*diff + (1-diff)*np.
˓→array(current_smiles[parent2])
child2 = np.array(current_smiles[parent1])*(1-diff)+ diff*np.
˓→array(current_smiles[parent2])

## Add child1
new_smiles = smile_to_smiles_percentage(np.array(child1) .
˓→reshape(-1,LATENT_DIM),1.0)
try:
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = []
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    if len(new_smiles_enc) != 0:
        ic50_preds = list(get_ic50s_mult(new_smiles_enc,gene_expression))
        new_ic50s.extend(ic50_preds)
        new_current_smiles.extend(list(new_smiles_enc))
        new_decoded_smiles.extend(cnew_decoded_smiles)
except:
    continue

## Add child2
new_smiles = smile_to_smiles_percentage(np.array(child2) .
˓→reshape(-1,LATENT_DIM),1.0)
try:
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = []
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    if len(new_smiles_enc) != 0:
        ic50_preds = list(get_ic50s_mult(new_smiles_enc,gene_expression))
        new_ic50s.extend(ic50_preds)
        new_current_smiles.extend(list(new_smiles_enc))
        new_decoded_smiles.extend(cnew_decoded_smiles)
except:
    continue
print("BEST CHILD: " +new_decoded_smiles[np.argmin(new_ic50s)])
print("BEST CHILD IC50: {:.5f}".format(min(new_ic50s))+'\n')

for smiles in current_smiles:
    #norm = np.random.uniform(0, jump_size)
    new_smiles = smile_to_smiles_percentage(np.array(smiles) .
˓→reshape(-1,LATENT_DIM),norm)

```

```

try:
    new_smiles = set(new_smiles)
    cnew_decoded_smiles = list([decoded_smiles[idx]])
    cnew_decoded_smiles.extend(new_smiles)
    new_smiles_enc = np.array(encode(new_smiles)).reshape(-1,LATENT_DIM)
    new_smiles_enc = np.concatenate((np.array(smiles).
    ↪reshape(-1,LATENT_DIM),new_smiles_enc))
    ic50_preds = list(get_ic50s_mult(new_smiles_enc,gene_expression))
    new_ic50s.extend(ic50_preds)
    new_current_smiles.extend(list(new_smiles_enc))
    new_decoded_smiles.extend(cnew_decoded_smiles)
except:
    continue
print("BEST CURRENT EVOLUTION: "+new_decoded_smiles[np.argmin(new_ic50s)])
print("BEST EVOLUTION IC50 : {:.5f}".format(min(new_ic50s))+'\n')
idx +=1
print('THIS GENERATION:')
decoded_smiles.extend(new_decoded_smiles)
current_smiles.extend(new_current_smiles)
top_ic50s.extend(new_ic50s)
top_ic50s, current_smiles, decoded_smiles = (list(t) for t in
    ↪zip(*sorted(zip(top_ic50s, np.array(current_smiles).tolist(), np.
    ↪array(decoded_smiles).tolist() ))))
non_repeat_decoded_smiles = []
indexes = []
index = 0
for smiles in decoded_smiles:
    if smiles not in non_repeat_decoded_smiles:
        non_repeat_decoded_smiles.append(smiles)
        indexes.append(index)
    index+=1
decoded_smiles = np.array(decoded_smiles)[indexes].tolist()
top_ic50s = np.array(top_ic50s)[indexes].tolist()
current_smiles = np.array(current_smiles)[indexes].tolist()
top_ic50s = top_ic50s[:num_out]
current_smiles = current_smiles[:num_out]
decoded_smiles = decoded_smiles[:num_out]
print("BEST GENERATION : "+ decoded_smiles[np.argmin(top_ic50s)])
print("BEST GENERATION IC50 : {:.5f}".format(min(top_ic50s)) +'\n')
print(decoded_smiles)

```

```
[ ]: ## Get necessary train approved cancer drugs for comparsion
train_smiles_path = './Datasets/train_Tox_data.smi'
smiles_train = pd.read_csv(train_smiles_path,delimiter='\t',header=None)
```

```
[ ]: ms = [rdkit.Chem.MolFromSmiles(x) for x in smiles_train[0]]
indexs = [i for i,v in enumerate(ms) if v != None]
ms =[v for i,v in enumerate(ms) if v != None]
fps_drugs = [rdkit.Chem.RDKFingerprint(x) for x in ms]

[ ]: ms = [rdkit.Chem.MolFromSmiles(x) for x in decoded_smiles]
indexs = [i for i,v in enumerate(ms) if v != None]
ms =[v for i,v in enumerate(ms) if v != None]
fps_cand = [rdkit.Chem.RDKFingerprint(x) for x in ms]

[ ]: from rdkit import DataStructs
closest_drugs = []
closest_values = []
for idx1 in range(len(fps_cand)):
    closest_drugs.append(smiles_train[0][0])
    closest_values.append(0)
    for idx2 in range(len(fps_drugs)):
        tanimoto = DataStructs.FingerprintSimilarity(fps_cand[idx1],fps_drugs[idx2])
        if closest_values[idx1] < tanimoto:
            closest_values[idx1] = tanimoto
            closest_drugs[idx1] = smiles_train[0][idx2]

[ ]: m
```

References

- [1] Jay Alammar. *The Illustrated GPT-2 (Visualizing Transformer Language Models) – Jay Alammar – Visualizing machine learning one concept at a time*. URL: <http://jalammar.github.io/illustrated-gpt2/> (visited on 08/15/2020).
- [2] Josep Arús-Pous et al. “Randomized SMILES Strings Improve the Quality of Molecular Generative Models”. In: (July 2019). DOI: 10.26434/CHEMRXIV.8639942.V2.
- [3] Tristan Aumentado-Armstrong. *Latent Molecular Optimization for Targeted Therapeutic Design*. Tech. rep. arXiv: 1809.02032v1.
- [4] Senem Aykul and Erik Martinez-Hackert. “Determination of half-maximal inhibitory concentration using biosensor-based protein interaction analysis”. In: *Analytical Biochemistry* 508 (Sept. 2016), pp. 97–103. ISSN: 10960309. DOI: 10.1016/j.ab.2016.06.025. URL: <https://pubmed.ncbi.nlm.nih.gov/27365221/>.
- [5] Pierre Baldi and Ramzi Nasr. “When is chemical similarity significant? the statistical distribution of chemical similarity scores and its extreme values”. In: *Journal of Chemical Information and Modeling* 50.7 (July 2010), pp. 1205–1222. ISSN: 15499596. DOI: 10.1021/ci100010v. URL: <https://pmc/articles/PMC2914517/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2914517/>.
- [6] A. Patrícia Bento et al. “The ChEMBL bioactivity database: An update”. In: *Nucleic Acids Research* 42.D1 (Jan. 2014), pp. D1083–D1090. ISSN: 03051048. DOI: 10.1093/nar/gkt1031. URL: <https://academic.oup.com/nar/article/42/D1/D1083/1043509>.
- [7] Sanjivanjit K Bhal. *Application Note LogP-Making Sense of the Value*. Tech. rep. URL: www.acdlabs.com.
- [8] Sanjit Bhat et al. “Var-CNN: A Data-Efficient Website Fingerprinting Attack Based on Deep Learning”. In: *Proceedings on Privacy Enhancing Technologies* 2019.4 (Feb. 2018), pp. 292–310. DOI: 10.2478/popets-2019-0070. arXiv: 1802.10215. URL: <http://arxiv.org/abs/1802.10215%20http://dx.doi.org/10.2478/popets-2019-0070>.
- [9] G. Richard Bickerton et al. “Quantifying the chemical beauty of drugs”. In: *Nature Chemistry* 4.2 (Feb. 2012), pp. 90–98. ISSN: 17554330. DOI: 10.1038/nchem.1243. URL: <https://pmc/articles/PMC3524573/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3524573/>.
- [10] Jannis Born et al. *PaccMann RL : Designing anticancer drugs from transcriptomic data via reinforcement learning*. Tech. rep. 2020. arXiv: 1909.05114v4.

- [11] Leo Breiman. “Random forests”. In: *Machine Learning* 45.1 (Oct. 2001), pp. 5–32. ISSN: 08856125. DOI: 10.1023/A:1010933404324. URL: <https://link.springer.com/article/10.1023/A:1010933404324>.
- [12] Nathan Brown et al. “GuacaMol: Benchmarking Models for de Novo Molecular Design”. In: (2019). DOI: 10.1021/acs.jcim.8b00839. URL: <https://benevolent.ai/guacamol..>
- [13] Yoosup Chang et al. “Cancer Drug Response Profile scan (CDRscan): A Deep Learning Model That Predicts Drug Effectiveness from Cancer Genomic Signature”. In: *Scientific Reports* 8.1 (Dec. 2018), p. 8857. ISSN: 20452322. DOI: 10.1038/s41598-018-27214-6. URL: <https://gdc.cancer..>
- [14] Inc Daylight Chemical Information Systems. *Daylight Theory: Fingerprints*. URL: <https://www.daylight.com/dayhtml/doc/theory/theory.finger.html> (visited on 09/12/2020).
- [15] Hans De Wolf et al. “High-Throughput Gene Expression Profiles to Define Drug Similarity and Predict Compound Activity”. In: *Assay and Drug Development Technologies* 16.3 (Apr. 2018), pp. 162–176. ISSN: 15578127. DOI: 10.1089/adt.2018.845. URL: <https://pubmed.ncbi.nlm.nih.gov/29658791/>.
- [16] Peter Ertl and Ansgar Schuffenhauer. “Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions”. In: *Journal of Cheminformatics* 1.1 (Dec. 2009), p. 8. ISSN: 17582946. DOI: 10.1186/1758-2946-1-8. URL: <https://jcheminf.biomedcentral.com/articles/10.1186/1758-2946-1-8>.
- [17] Le Fang et al. *Implicit Deep Latent Variable Models for Text Generation*. Tech. rep., pp. 3946–3956. URL: <https://github.com/fangleai/>.
- [18] Paulo Fernandes, João Correia, and Penousal Machado. *Evolutionary Latent Space Exploration of Generative Adversarial Networks*. Tech. rep.
- [19] Peter I Frazier. *A Tutorial on Bayesian Optimization*. Tech. rep. 2018. arXiv: 1807.02811v1.
- [20] Hao Fu et al. *Cyclical Annealing Schedule: A Simple Approach to Mitigating KL Vanishing*. Tech. rep. arXiv: 1903.10145v3.
- [21] Paul Geleher, Nancy J. Cox, and R. Stephanie Huang. “Cancer biomarker discovery is improved by accounting for variability in general levels of drug sensitivity in pre-clinical models”. In: *Genome Biology* 17.1 (Sept. 2016), p. 190. ISSN: 1474760X. DOI: 10.1186/s13059-016-1050-9. URL: <https://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1050-9>.
- [22] Pierre Geurts, Damien Ernst, and Louis Wehenkel. “Extremely randomized trees”. In: *Machine Learning* 63.1 (Apr. 2006), pp. 3–42. ISSN: 08856125. DOI: 10.1007/s10994-006-6226-1.
- [23] Philip D. Glaves and Jonathan D. Tugwood. “Generation and analysis of transcriptomics data.” In: *Methods in molecular biology (Clifton, N.J.)* 691 (2011), pp. 167–185. ISSN: 19406029. DOI: 10.1007/978-1-60761-849-2_10. URL: <https://pubmed.ncbi.nlm.nih.gov/20972753/>.

- [24] Rafael Gómez-Bombarelli et al. “Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules”. In: *ACS Central Science* 4.2 (Feb. 2018), pp. 268–276. ISSN: 23747951. DOI: 10.1021/acscentsci.7b00572. arXiv: 1610.02415. URL: <https://pubs.acs.org/sharingguidelines>.
- [25] Daria Grechishnikova. “Transformer neural network for protein specific de novo drug generation as machine translation problem”. In: *bioRxiv* (Dec. 2019), p. 863415. DOI: 10.1101/863415. URL: <https://doi.org/10.1101/863415>.
- [26] Ryan Rhys Griffiths and José Miguel Hernández-Lobato. “Constrained Bayesian optimization for automatic chemical design using variational autoencoders”. In: *Chemical Science* 11.2 (Jan. 2020), pp. 577–586. ISSN: 20416539. DOI: 10.1039/c9sc04026a. URL: <https://github.com/Ryan-Rhys/Constrained->.
- [27] Ryan-Rhys Griffiths and José Miguel Hernández-Lobato. *Constrained Bayesian Optimization for Automatic Chemical Design*. Tech. rep. arXiv: 1709.05501v6.
- [28] Francesco Iorio et al. “A Landscape of Pharmacogenomic Interactions in Cancer”. In: *Cell* 166.3 (July 2016), pp. 740–754. ISSN: 10974172. DOI: 10.1016/j.cell.2016.06.017. URL: <https://pubmed.ncbi.nlm.nih.gov/27397505/>.
- [29] John J. Irwin and Brian K. Shoichet. “ZINC – A Free Database of Commercially Available Compounds for Virtual Screening”. In: *Journal of chemical information and modeling* 45.1 (2005), p. 177. DOI: 10.1021/CI049714. URL: [/pmc/articles/PMC1360656/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1360656/](https://pmc/articles/PMC1360656/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1360656/).
- [30] Łukasz Kaiser et al. *Fast Decoding in Sequence Models Using Discrete Latent Variables*. Tech. rep. 2018. arXiv: 1803.03382v6.
- [31] Minoru Kanehisa et al. “KEGG for representation and analysis of molecular networks involving diseases and drugs”. In: *Nucleic Acids Research* 38.SUPPL.1 (Oct. 2009). ISSN: 03051048. DOI: 10.1093/nar/gkp896. URL: <https://pubmed.ncbi.nlm.nih.gov/19880382/>.
- [32] Sunghwan Kim et al. “PubChem 2019 update: Improved access to chemical data”. In: *Nucleic Acids Research* 47.D1 (Jan. 2019), pp. D1102–D1109. ISSN: 13624962. DOI: 10.1093/nar/gky1033. URL: <http://apps.who.int/>.
- [33] Antje Kirchner and Curtis S. Signorino. “Using Support Vector Machines for Survey Research”. In: *Survey Practice* 11.1 (Jan. 2018), pp. 1–14. ISSN: 2168-0094. DOI: 10.29115/sp-2018-0001.
- [34] Mario Krenn et al. *Self-Referencing Embedded Strings (SELFIES): A 100% robust molecular string representation*. Tech. rep. arXiv: 1905.13741v2. URL: <https://github.com/>.
- [35] Emilia Lim et al. “T3DB: A comprehensively annotated database of common toxins and their targets”. In: *Nucleic Acids Research* 38.SUPPL.1 (Nov. 2009), p. D781. ISSN: 03051048. DOI: 10.1093/nar/gkp934. URL: [/pmc/articles/PMC2808899/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2808899/](https://pmc/articles/PMC2808899/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2808899/).
- [36] Ann Lin et al. “Off-target toxicity is a common mechanism of action of cancer drugs undergoing clinical trials”. In: *Science Translational Medicine* 11.509 (Sept. 2019). ISSN: 19466242. DOI: 10.1126/scitranslmed.aaw8412. URL: <https://pubmed.ncbi.nlm.nih.gov/31511426/>.

- [37] Danyang Liu and Gongshen Liu. “A Transformer-Based Variational Autoencoder for Sentence Generation”. In: *Proceedings of the International Joint Conference on Neural Networks*. Vol. 2019-July. Institute of Electrical and Electronics Engineers Inc., July 2019. ISBN: 9781728119854. DOI: 10.1109/IJCNN.2019.8852155.
- [38] Ian Lloyd. *Pharma R&D Annual Review 2018*. Tech. rep. 2018.
- [39] Rohan Lowe et al. “Transcriptomics technologies”. In: *PLoS Computational Biology* 13.5 (May 2017), e1005457. ISSN: 15537358. DOI: 10.1371/journal.pcbi.1005457. URL: <https://doi.org/10.1371/journal.pcbi.1005457.t001>.
- [40] James Lucas et al. *Understanding Posterior Collapse in Generative Latent Variable Models*. Tech. rep.
- [41] Matteo Manica et al. *Towards Explainable Anticancer Compound Sensitivity Prediction via Multimodal Attention-based Convolutional Encoders*. Tech. rep. 2019. arXiv: 1904.11223v3.
- [42] Manik Soni. *Understanding architecture of LSTM cell from scratch with code*. | Hacker Noon. URL: <https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4> (visited on 08/16/2020).
- [43] Oscar Méndez-Lucio et al. “De novo generation of hit-like molecules from gene expression signatures using artificial intelligence”. In: *Nature Communications* 11.1 (Dec. 2020), pp. 1–10. ISSN: 20411723. DOI: 10.1038/s41467-019-13807-w. URL: <https://doi.org/10.1038/s41467-019-13807-w>.
- [44] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. Tech. rep.
- [45] National Institutes of Health U.S. National Library of Medicine. *Amino acids: MedlinePlus Medical Encyclopedia*. URL: <https://medlineplus.gov/ency/article/002222.htm> (visited on 09/06/2020).
- [46] Noel O’boyle and Andrew Dalke. “DeepSMILES: An Adaptation of SMILES for Use in Machine-Learning of Chemical Structures”. In: (). DOI: 10.26434/chemrxiv.7097960.v1. URL: <https://github.com/nextmovesoftware/deepsmiles>.
- [47] Marek Obitko. *Crossover and mutation - Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets*. URL: <https://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php> (visited on 08/26/2020).
- [48] P. G. Polishchuk, T. I. Madzhidov, and A. Varnek. “Estimation of the size of drug-like chemical space based on GDB-17 data”. In: *Journal of Computer-Aided Molecular Design* 27.8 (Aug. 2013), pp. 675–679. ISSN: 0920654X. DOI: 10.1007/s10822-013-9672-4. URL: <https://pubmed.ncbi.nlm.nih.gov/23963658/>.
- [49] Limeng Pu et al. “EToxPred: A machine learning-based approach to estimate the toxicity of drug candidates 11 Medical and Health Sciences 1115 Pharmacology and Pharmaceutical Sciences 03 Chemical Sciences 0305 Organic Chemistry 03 Chemical Sciences 0304 Medicinal and Biomol”. In: *BMC Pharmacology and Toxicology* 20.1 (Jan. 2019), p. 2. ISSN: 20506511. DOI: 10.1186/s40360-018-0282-6. URL: <https://bmcparmacoltoxicol.biomedcentral.com/articles/10.1186/s40360-018-0282-6>.

- [50] Alec Radford et al. *Language Models are Unsupervised Multitask Learners*. Tech. rep. URL: <https://github.com/codelucas/newspaper>.
- [51] K. V. Rashmi and Ran Gilad-Bachrach. “DART: Dropouts meet Multiple Additive Regression Trees”. In: *Journal of Machine Learning Research* 38 (May 2015), pp. 489–497. arXiv: 1505.01866. URL: <http://arxiv.org/abs/1505.01866>.
- [52] Anwar Rayan, Jamal Raiyn, and Mizied Falah. “Nature is the best source of anticancer drugs: Indexing natural products for their anticancer bioactivity”. In: *PLOS ONE* 12.11 (Nov. 2017). Ed. by Irina V Lebedeva, e0187925. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0187925. URL: <https://dx.plos.org/10.1371/journal.pone.0187925>.
- [53] RDKIT. *The RDKit Book — The RDKit 2020.03.1 documentation*. URL: https://www.rdkit.org/docs/RDKit%7B%5C_%7DBook.html (visited on 08/31/2020).
- [54] R. T. Rockafellar. *Extension of fenchels duality Theorem for convex functions*. 1966. DOI: 10.1215/S0012-7094-66-03312-6. URL: <https://projecteuclid.org/euclid.dmj/1077376222>.
- [55] David Rogers and Mathew Hahn. “Extended-connectivity fingerprints”. In: *Journal of Chemical Information and Modeling* 50.5 (May 2010), pp. 742–754. ISSN: 15499596. DOI: 10.1021/ci100050t. URL: <https://pubs.acs.org/doi/full/10.1021/ci100050t>.
- [56] Jack W. Scannell et al. *Diagnosing the decline in pharmaceutical R&D efficiency*. Mar. 2012. DOI: 10.1038/nrd3681. URL: <https://pubmed.ncbi.nlm.nih.gov/22378269/>.
- [57] Gisbert Schneider. “Mind and machine in drug design”. In: *Nature Machine Intelligence* 1.3 (Mar. 2019), pp. 128–130. ISSN: 25225839. DOI: 10.1038/s42256-019-0030-7. URL: <https://www.nature.com/articles/s42256-019-0030-7>.
- [58] SMILES. *Daylight Theory: SMILES*. URL: <https://www.daylight.com/dayhtml/doc/theory/theory.smiles.html> (visited on 08/03/2020).
- [59] Gail Spooner. *Activation Functions*. URL: <http://gailsponner.co.uk/bjrazb/prelu-activation-function.html> (visited on 09/09/2020).
- [60] Nitish Srivastava et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Tech. rep. 56. 2014, pp. 1929–1958. URL: <http://jmlr.org/papers/v15/srivastava14a.html>.
- [61] J. A.K. Suykens and J. Vandewalle. “Least squares support vector machine classifiers”. In: *Neural Processing Letters* 9.3 (1999), pp. 293–300. ISSN: 13704621. DOI: 10.1023/A:1018628609742. URL: <https://link.springer.com/article/10.1023/A:1018628609742>.
- [62] Damian Szkłarczyk et al. “STRING v10: Protein-protein interaction networks, integrated over the tree of life”. In: *Nucleic Acids Research* 43.D1 (Jan. 2015), pp. D447–D452. ISSN: 13624962. DOI: 10.1093/nar/gku1003. URL: <https://pubmed.ncbi.nlm.nih.gov/25352553/>.

- [63] Transcriptome. *Transcriptome Fact Sheet*. URL: <https://www.genome.gov/about-genomics/fact-sheets/Transcriptome-Fact-Sheet> (visited on 08/11/2020).
- [64] Ashish Vaswani et al. *Attention Is All You Need*. Tech. rep. arXiv: 1706.03762v5.
- [65] Bie Verbist et al. *Using transcriptomics to guide lead optimization in drug discovery projects: Lessons learned from the QSTAR project*. May 2015. DOI: 10.1016/j.drudis.2014.12.014. URL: <http://dx.doi.org/10.1016/j.drudis.2014.12.014>.
- [66] Pan Wang et al. “Datanet: Deep learning based encrypted network traffic classification in SDN home gateway”. In: *IEEE Access* 6 (2018), pp. 55380–55391. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2872430.
- [67] Martin Wehling. “Assessing the translatability of drug projects: what needs to be scored to predict success?” In: *Nature Reviews Drug Discovery* 8.7 (July 2009), pp. 541–546. ISSN: 1474-1776. DOI: 10.1038/nrd2898. URL: <http://www.nature.com/articles/nrd2898>.
- [68] Philip Wexler. “TOXNET: An evolving web resource for toxicology and environmental health information”. In: *Toxicology* 157.1-2 (Jan. 2001), pp. 3–10. ISSN: 0300483X. DOI: 10.1016/S0300-483X(00)00337-1. URL: <https://pubmed.ncbi.nlm.nih.gov/11164971/>.
- [69] David S. Wishart et al. “DrugBank: a comprehensive resource for in silico drug discovery and exploration.” In: *Nucleic acids research* 34.Database issue (2006). ISSN: 13624962. DOI: 10.1093/nar/gkj067. URL: <https://pubmed.ncbi.nlm.nih.gov/16381955/>.
- [70] Yu Fang Zhang et al. “SPVec: A Word2vec-Inspired Feature Representation Method for Drug-Target Interaction Prediction”. In: *Frontiers in Chemistry* 7 (Jan. 2020), p. 895. ISSN: 22962646. DOI: 10.3389/fchem.2019.00895. URL: [/pmc/articles/PMC6967417/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6967417/](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6967417/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6967417/).
- [71] Tiancheng Zhao, Kyusong Lee, and Maxine Eskenazi. *Unsupervised Discrete Sentence Representation Learning for Interpretable Neural Dialog Generation*. Tech. rep., pp. 1098–1107. URL: <https://github...>