3.1 Problem Understanding

In chapter 2, we identified hits with high similarity to one or more Malaria Box compounds. Our next task will be to develop a ligand-based virtual screening model to identify hits that pose cardiotoxic risk. Our model is just one component of a much bigger virtual screening pipeline. A perfectly accurate model will be useless if it doesn't fulfill the company's expected use cases and end goal, which leads us to the first question we need to ask: what are the project requirements, constraints, and goals?

As input, we will use our file containing compounds that have high similarity to at least one of the Malaria Box compounds. Though these compounds are promising hits, we want to further filter out any compounds that are active to a cardiotoxicity antitarget. An *antitarget* (or off-target) is a receptor, enzyme, or other biological target that, when affected by a drug, causes undesirable side-effects. Whereas we want to identify compounds that are active against a target, we want to remove compounds that are active against an antitarget. Antitargets are commonly biomolecules that play important roles in normal physiological processes that are not directly related to the condition that we want to treat.

The antitarget we'll consider is the human ether-a-go-go-related gene (hERG) potassium channel. Drug-induced blockage of the hERG channel is considered the primary cause of cardiotoxicity and must be screened for early in the drug discovery process. The expected output of your model is binary – the compound does not block the hERG channel and should be kept or it does block the hERG channel and should be removed. Furthermore, the company cares about two failure cases: (1) we miss a cardiotoxic compound that proceeds through the pipeline to clinical trials and fails (incurring cost and damaging brand reputation) and (2) we incorrectly classify an otherwise therapeutically beneficial compound as cardiotoxic and cut its candidacy short.

Coming to a second question to consider, how will we benchmark the success of our model? We could compare our model's performance against other known models, as well as methods that don't involve ML, such as rule-based filters to detect HERG blockage by proxy. For instance, blockage of the hERG channel is associated with long QT syndrome, which is a heart disorder that can cause arrhythmia (fast, rapid heartbeats). To estimate hERG blockage, AI we might construct a small set of structural alerts that are associated with arrhythmia endpoints. If a compound contains several structural alerts greater than a user-defined threshold of these structures, it is filtered out.

TANGENT **QT in "long QT" is a reference to the heart's electrical activity as recorded with an electrocardiogram (ECG). Different waves on the ECG graph are marked as P, Q, R, S, and T. Activity for Q through T corresponds to heart cells electrically "recharging" after muscle contraction. Long QT indicates slow recharging, which is associated with heart abnormalities.**

3.1.1 Your Machine Learning Task

Where to begin? What ML model to use? Why do we expect an ML model to be better than a simple rule-based approach like structural alerts? Furthermore, suppose we did know the target's structure. Why might we still think that machine learning can add value, rather than moving directly to a structure-based method? There are a few guiding principles for assessing when machine learning may be of value:

1. Existing data is available or can be collected.
2. The data contains complex patterns, and our model has the capacity to learn how those patterns correspond to known outcomes. Here, *capacity* refers to the model's ability to learn a variety of possible functions, where greater model complexity implies greater capacity in learning.
3. We have enough data that important patterns may be repeated multiple times, making it easier for the model to remember them.
4. Our model can solve problems by using what it has learned to make predictions on new data, assuming that the new data is similar to the model's training data.
5. Our model can be applied to new data at scale and speed that is not possible with current solutions.

Our task is a typical classification task: the model needs to output which class each compound belongs. A compound belongs to either the positive class if hERG blocking or negative class if not. Since there are only two classes to predict, this is a binary classification task. If we were trying to predict more than two possible classes, it would be a multiclass classification problem. In general, binary classification is easier than multiclass classification. Lastly, we can train a model using a labeled dataset of compounds known to either block or not block the hERG channel, which makes this a supervised learning task.

3.2 Data Acquisition, Exploration, & Curation

As we've detailed for reference in appendix B, there are many public databases such as PubChem and ChEMBL with existing data for molecular property prediction. Another valuable source of data are publications that make the data used in their experiments publicly available. If you are already familiar with a specific subfield in cheminformatics, then you likely know of common benchmark datasets derived from individual publications. If you are not familiar with a field and what dataset publications exist, it is helpful to reference a dataset aggregator. For example, Papers With Code open sources datasets across a variety of machine learning fields and tasks, including some pertaining to drug discovery. Another collection specifically for drug discovery is aggregated by Therapeutics Data Commons (TDC) [1].

To run the code examples through this chapter, navigate to https://github.com/nrflynn2/ml-drug-discovery. This leads to a GitHub repository with the full list of available Jupyter notebooks for each chapter of this book. To follow along for chapter 3, open the chapter labeled *CH03_Flynn_ML4DD.ipynb*.

3.2.1 Loading and Exploring the hERG Blockers Dataset

If we navigate to TDC datasets for toxicity, we can find references for three datasets related to hERG blockers. For ease of use, let's start off with downloading and loading the "hERG blockers" dataset into a Pandas data frame.

Listing 3.1 Loading hERG Blockers Dataset

```
from pathlib import Path
import pandas as pd
import urllib.request

def load_herg_blockers_data():
  herg_blockers_path = Path("datasets/herg_blockers.xlsx")

  if not herg_blockers_path.is_file():
    Path("datasets").mkdir(parents=True, exist_ok=True)
    url = " https://github.com/nrflynn2/ml-drug-
discovery/blob/main/data/hERG_blockers.xlsx"
    urllib.request.urlretrieve(url, herg_blockers_path)

  return pd.read_excel(
      herg_blockers_path,
      usecols="A:F",
      header=None,
      skiprows=[0,1],
      names=["SMILES", "Name", "pIC50", "Class", "Scaffold Split", "Random Split"],
    ).head(-68)                                                    #A
```

#A We remove the last 68 rows, which refer to an evaluation set used by the paper's authors for a different experiment.

UNDERSTANDING THE DATA STRUCTURE

We can peek at the top three rows via herg_blockers.head(3)(table 3.1).

Table 3.1 The first three rows of the hERG blockers dataset.

SMILES	Name	pIC50	Class	Scaffold Split	Random Split
Fc1ccc(cc1)Cn1c2c(nc1NC1CCN(CC1)CCc1ccc(O)cc1)cccc2	DEMETHYLASTEMIZOLE	9.0	1.0	Training I	Training II
Fc1ccc(cc1)C(OCC[NH+]1CC[NH+](CC1)CCCc1ccccc1)c1ccc(F)cc1	GBR-12909	9.0	1.0	Training I	Training II
O=[N+]([O-])c1ccc(cc1)CCCCN(CCCCCCC)CC	LY-97241	8.8	1.0	Training I	Training II

Let's take a quick look at the non-null count and data type of each column:

Listing 3.2 View total number of rows, number of non-null values, and data types

```
>>> herg_blockers.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 587 entries, 0 to 586
Data columns (total 6 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   SMILES          587 non-null    object
 1   Name            565 non-null    object
 2   pIC50           526 non-null    float64
 3   Class           587 non-null    float64
 4   Scaffold Split  587 non-null    object
 5   Random Split    587 non-null    object
dtypes: float64(2), object(4)
memory usage: 27.6+ KB
```

The hERG blockers data is a set of 587 compounds. Each row represents one compound. There are six columns:

- Name: The name of the compound.
- SMILES: Textual representation of each compound's structure.
- pIC_{50}: Higher pIC_{50} indicates greater drug potency. pIC_{50} is derived from IC_{50}, an experimental measure of the concentration of a drug at which it inhibits a specific biological process by 50%. The "IC" stands for *inhibitory concentration.* Reported IC_{50} values can have a large range and different units. To standardize comparison, IC_{50} is often converted to pIC_{50}. pIC_{50} is the negative of the log of the IC_{50} in molar (M). Molar (M) is a measurement of the concentration of a chemical species. For example:
  - An IC_{50} of 1 nM is 10^{-9} M, which is a pIC_{50} of 9.0. An IC_{50} of 1 µM is 10^{-6} M, which is a pIC_{50} of 6.0.
  - This attribute is null for several compounds, but we can't use this attribute for training and it did not affect the number of non-null class attributes that we will use for labels.
- Class: Each drug is labeled with a "1" if it blocks the hERG channel and a "0" if it does not block the hERG channel. This is the column that we want to train a model to be able to predict. The authors used a threshold of $IC_{50} < 40$ µM to define drugs as hERG blockers. This threshold corresponds to a $pIC_{50} > 4.4$.
- Set I: The authors split the dataset into training and test data using a scaffold split. We won't use the set I column until we cover scaffold splitting later.

- Set II: The authors split the dataset into training and test data using a random split. We will use the set II column to define our training and test data.

There is a clear demarcation between the compound classes based on pIC_{50}. However, we can't use the pIC_{50} column as a feature for model training as pIC_{50} was used to define the class variable we are predicting. Furthermore, we may not want to rely on pIC50 measurements as it requires experimental data, which may be costly, and pIC50 data quality may vary due to missing values or inconsistent experiment conditions or readouts. Furthermore, pIC_{50} is experimentally derived. The purpose of our model is to predict whether a compound is a hERG blocker or not prior to conducting an experiment.

Regardless of whether the data comes from a large, well-established public or commercial database or a small dataset in literature, curation is always necessary to mitigate possible mistakes in the dataset. Manually reviewing all dataset entries is not feasible and even a random selection of a small subset, while useful, might miss problematic entries. Instead, we can harness exploratory data analysis (EDA) to plot and summarize features of the data. EDA helps us understand the range of molecular properties that the data covers and can be used to spot inconsistencies.

VERIFYING THE TARGET PROPERTY DISTRIBUTION

Let's get familiar with our data by plotting the distribution of pIC_{50} values:

Listing 3.3 Distribution of pIC_{50} for real dataset and dataset with simulated error

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.histplot(
    herg_blockers["pIC50"], kde=True,
    stat="density", kde_kws=dict(cut=3),
    alpha=.4, edgecolor=(1, 1, 1, .4),
)
plt.title("Distribution of pIC50")

simulated_error = herg_blockers["pIC50"] + 3.0                    #A
sns.histplot(
    herg_blockers["pIC50"].append(simulated_error, ignore_index=True), kde=True,
    stat="density", kde_kws=dict(cut=3),
    alpha=.4, edgecolor=(1, 1, 1, .4),
)
plt.title("Distribution of pIC50, Simulated Annotation Error")
```

#A Shifting the pIC50 distribution to simulate annotation error due to units disagreement.

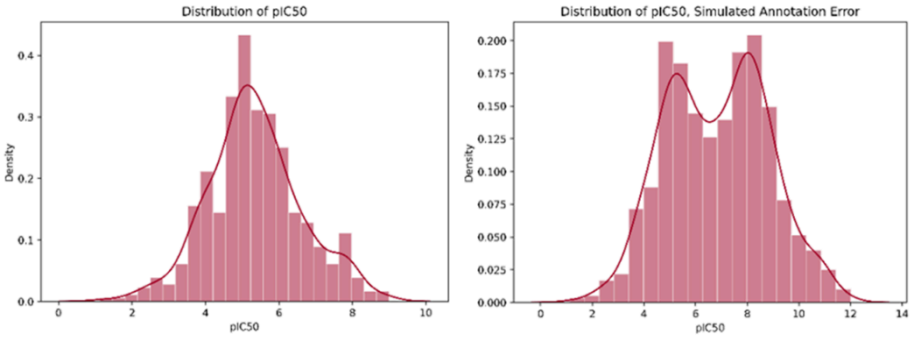

Figure 3.2 (Left) Distribution of pIC_{50} values for the real hERG blockers dataset. (Right) Distribution of pIC_{50} values for hERG blockers dataset with simulated curation error.

Reviewing the distribution of continuous variables can be useful to ensure that the range of values falls within what we would expect (figure 3.2). Even though we won't use pIC_{50} values for modeling, it's possible that there were annotation errors when logging the experimental pIC_{50} values. For example, if the distribution of pIC_{50} values followed a bimodal distribution 3 units apart as shown in figure 3.2, then we may want to dive deeper into what could be a potential data curation issue (i.e., the annotator used units of nanomolar instead of micromolar).

3.2.2 Validating & Standardizing SMILES

Let's visualize a few of the drugs in this dataset, both to understand the diversity in structure across blockers and non-blockers and to get a feel for how SMILES corresponds to the actual drug structure. In addition, this can help us to spot errors in drug structure that are more likely at extremes of the data.

Listing 3.4 Visualizing outlier structures

```
extremes = pd.concat([herg_blockers[:4], herg_blockers.dropna()[-4:]])    #A
legend_text = [
    f"{x.Name}: pIC50 = {x.pIC50:.2f}" for x in extremes.itertuples()
]
extreme_mols = [rdkit.Chem.MolFromSmiles(smi) for smi in extremes.SMILES]
rdkit.Chem.Draw.MolsToGridImage(
    extreme_mols, molsPerRow=4, subImgSize=(250, 250), legends=legend_text
)
```

#A Note that the data set we are using is already ordered by pIC50 values, so we can directly extract the extremes at each end of the data.

From figure 3.3, we can see some shared structural patterns between LY-97241 and clofilium phosphate, which are hERG blockers, and between sertindole16 and sertindole5, which are not hERG blockers. If we were to review more of the compounds, we'd see that these structural patterns can be complex but could be exploited by ML. We posit that our ML model can *learn* an optimal function for detecting structural patterns linked to hERG blockage, rather than us retrospectively determining structural alerts from manual inspection.

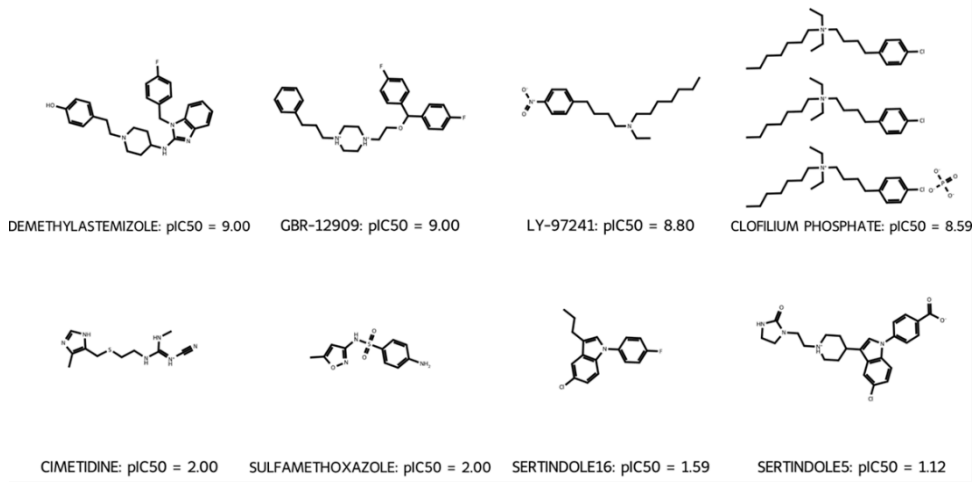

Figure 3.3 Top row displays the structures of the compounds with highest experimentally measured pIC_{50}. Bottom row displays the structures of the compounds with lowest experimentally measured pIC_{50}.

Inspection at the extremes is also more likely to catch outliers or mistakes. Should clofilium phosphate have repeated structures or is that a mistake? Should GBR-12909 be charged considering that other references to its structure in literature report the structure as uncharged?

To address these problems, we standardize the data. Standardized data allows for meaningful comparisons between different experiments or studies that aid in reproducibility of experiments, sharing data collected from various sources, and dissemination of results.In contrast to molecular validation, molecular standardization *modifies* the compounds with a set of standard operations that align the representations of all compounds in our dataset (figure 3.4).

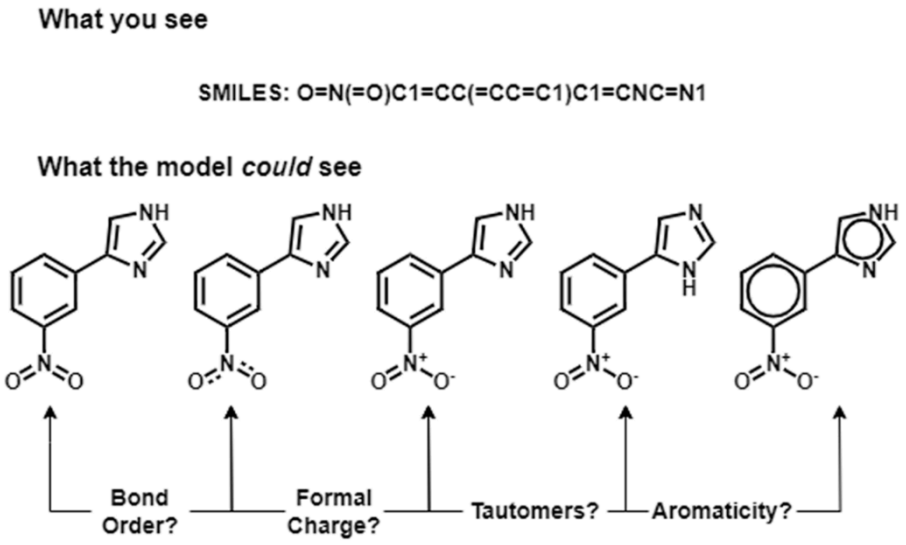

Figure 3.4 Our model does not see molecules in the same way we do. SMILES are one method of storage information about chemical structures. Standardization streamlines the different views our model could have of a molecule into one well-defined representation and aligns that well-defined representation across all molecules in the dataset. Prior to standardization, each molecule could have been curated dataset following different conventions that confuse and alter how our model views them, negatively influence its learning.

We will use a standardization pipeline that chains the following processes supported within RDKit's rdMolStandardize module. Listing 4.3 provides the code for constructing our standardization pipeline, and figure 3.5 visualizes example structures before and after standardization:

1. Disconnect metal atoms that are covalently bonded to non-metal atoms.
2. Normalize the molecules to cleanup functional groups.
3. Assign stereochemistry and remove redundant chirality specifications.
4. Remove salts, solvents, and other fragments, retaining the largest parent fragment.
5. Neutralize the molecule by adding or removing hydrogens where possible.
6. Enumerate all possible tautomers, score them, and retain a canonical tautomer based on the assigned scores (e.g., the most stable tautomer).

NOTE Due to the context-dependent nature of molecular standardization, we only mention a common set of operations. There are additional task-specific operations we don't mention because they aren't relevant to this chapter. For example, prior to docking we might charge molecules to represent how they would appear at physiological pH. For broader and more detailed coverage of molecular standardization operations, we recommend Greg Landrum's workshop on chemical structure validation and standardization [2].

Listing 3.5 SMILES to Mol Conversion & Mol Standardization

```
from rdkit.Chem.MolStandardize.rdMolStandardize import (
    Cleanup,
    LargestFragmentChooser,
    TautomerEnumerator,
    Uncharger
)

def process_smiles(smi):
  mol = rdkit.Chem.MolFromSmiles(smi)
  mol = Cleanup(mol)                                          #A
  mol = LargestFragmentChooser().choose(mol)                  #B
  mol = Uncharger().uncharge(mol)                             #C
  return TautomerEnumerator().Canonicalize(mol)               #D
```

#A Sensible defaults to standardize a molecule.
#B Retain the largest parent fragment.
#C Neutralize the molecule.
#D Find and retain a canonical tautomer representation of the molecule.

```
herg_blockers["mol"] = herg_blockers["SMILES"].apply(process_smiles)
```

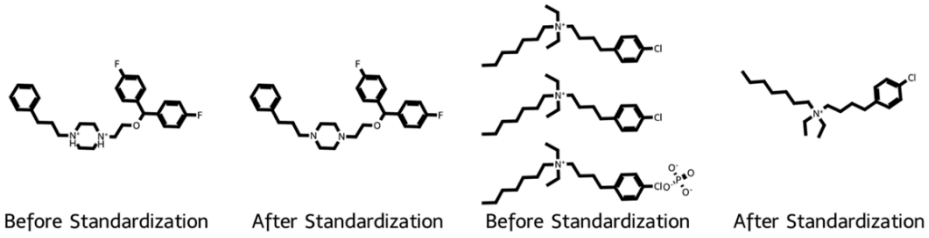

Figure 3.5 After standardization, GBR-12909 has been neutralized and clofilium phosphate has had excess fragments removed. Our next step will be to represent the dataset as fingerprints, which doesn't make sense for entries that would be more than one fragment.

3.2.3 Feature Generation & Exploration

Now that we have standardized our data set to a consistent format, we'll compute 2048-dimensional representations of the 587 molecules using Morgan fingerprints, as introduced in chapter 2 with code provided in listing 3.6. Once we've represented our data in terms of fingerprints, we can begin to visualize the distribution of the bits set across all fingerprints to try and understand more about the diversity of structures that might be represented.

Listing 3.6 Featurizing molecules with Morgan fingerprints

```
from rdkit import DataStructs
from rdkit.Chem import AllChem

def compute_fingerprint(mol: rdkit.Chem.Mol, r: int, nBits: int) -> np.ndarray:
    fp = AllChem.GetMorganFingerprintAsBitVect(mol, r, nBits=nBits)
    arr = np.zeros((1,), dtype=np.int8)
    DataStructs.ConvertToNumpyArray(fp, arr)
    return arr

fingerprints = np.stack([compute_fingerprint(mol, 2, 2048) for mol in
herg_blockers.mol])
```

Our hERG blockers dataset contains new fingerprint descriptors that we can analyze prior to modeling. From figure 3.6 (left), the data's fingerprint representation is sparse, though there is noticeable fluctuation in the frequency at which some bits are set, and the number of bits set by each molecule. Most bits are set by 30 or less molecules out of the possible 587 compounds. However, we can see several outlier bits that are set by hundreds of compounds. These outlier bits likely correspond to common structures present in most of the compounds in our data.

Figure 3.6 (right) gives a clearer picture of the distribution of bits set per molecule. Each compound typically sets a spread of 40 to 60 bits. At the extremes, there are compounds with as little as approximately 10 bits set as well as compounds that max out at 100 bits set out of the possible 2048 bits.

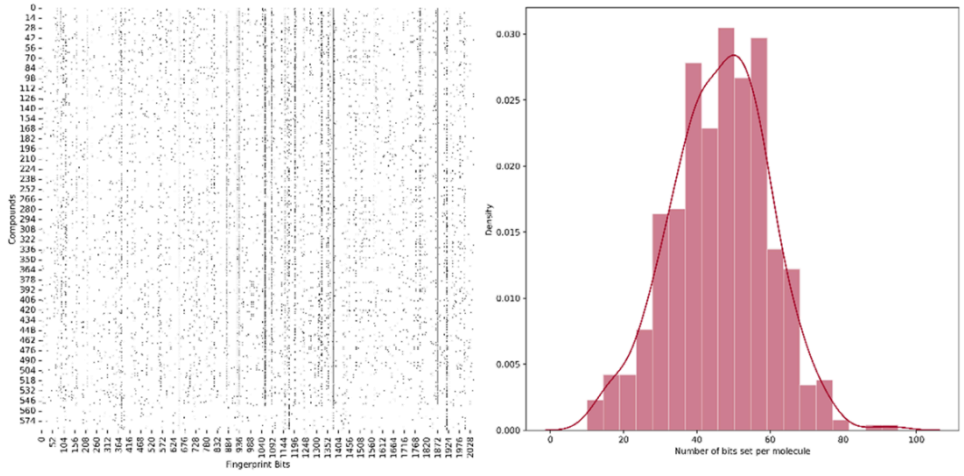

Figure 3.6 Molecules to fingerprint bits share a many-to-many relationship. (Left) Heatmap showing sparsity of set fingerprint bits across the entire training data set. (Right) The distribution of how many bits are set per molecule in the dataset.

With fingerprints to provide a numerical representation of each molecule, we can begin thinking about how we can leverage this representation for a machine learning model to establish a relationship between fingerprint bits and their influence on hERG blocking predisposition.

3.3 Application of Linear Models

We now enter what is typically viewed as the "fun" part of our ML workflow – modeling. During training, our ML model ingests data encoded as features and learns how to map those features to the property we want to predict. Before we train our linear model, we'll formalize what we want our model to achieve and how to do so via the data available to it.

3.3.1 Learning from Data

A medicinal chemist might be able to look at a chemical compound and recognize whether it contains certain substructures that are likely to cause hERG blockage. How do we get a computer to do the same? We leverage extensive, historical data on past compounds, which are labeled to indicate whether they were positive or negative for hERG blockage according to previous experiments. This labeled data set forms the basis of what our linear, supervised model will learn from.

DATA GENERATING DISTRIBUTION

We can formally define our problem as a mathematical function, f X Y, that maps the set of all possible inputs, X, to the set of all possible outputs, Y. In this case, the set of all possible inputs is the full chemical space of druglike compounds, and the set of all possible outputs is a simple "yes" or "no" determination about whether an input compound will be hERG blocking. Each individual compound, x, is represented with feature information that might aid in predicting whether it has "anti-cancer" behavior. Our data set contains pairs of input-output examples $(x_1,y_1),\ldots,(x_N,y_N)$, where each y_n for n=1,...,N is the output of feeding its corresponding input, x_n, into f.

The function *f* represents the solution to our problem, and we refer to it as the **target function**. The target function represents ground truth. If we have *f*, then we can determine whether any compound is hERG blocking. But if we already knew exactly what f is, then we wouldn't need to train a model to begin with!

Sadly, f is unknown to us and we must try to approximate the entire function from the few instances that make up our training data set. We train a model that maps our inputs to a predicted output. Ideally, our trained model results in correct predictions on our training data set of historical compound behavior, while generalizing to new compounds that we *don't* have any training samples for and that the model has never encountered.

Figure 3.7 shows how this can be a leap of faith – the training data set represents a sampled snapshot of the entire space of drug-like chemical compounds. Depending on how the data was sourced, the subset of the chemical space that our model is exposed to could vary drastically. We hope that our trained model is correct where we don't have training samples and that the distribution of the training set approximates the ground truth distribution of the target function. The more data available to us, the more likely that both assumptions will hold and that we can achieve a performative model.

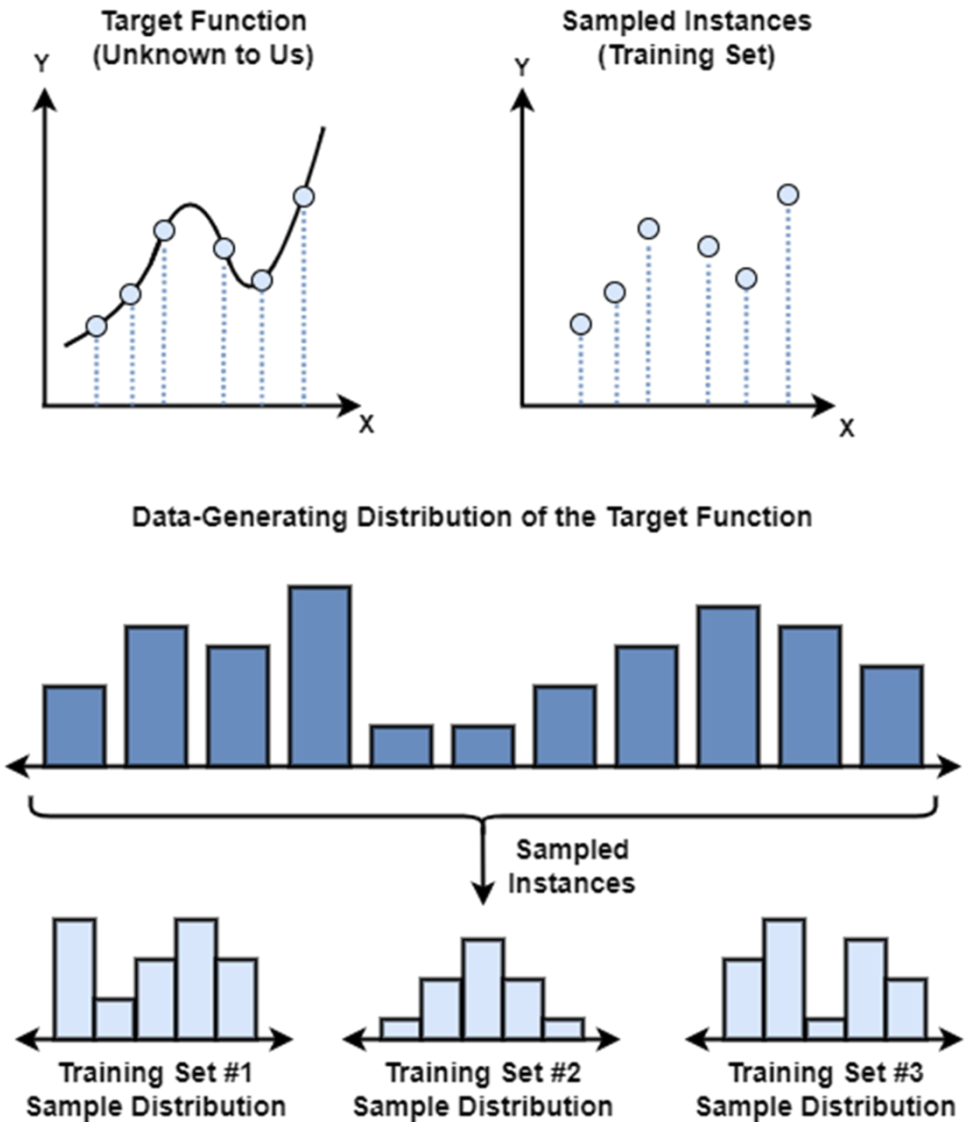

Figure 3.7 The target function represents ground truth. It is an unknown function that maps input data to their corresponding target or label values. The difficulty is that we must *learn the entire function* from a few training instances. We will estimate model weights that "fit" the training points exactly, assuming that the resulting function is correct where we *don't* have training instances. We also assume that the distribution of sampled data that constitutes our training set is equivalent to the target function's distribution. The more data available to us, the more likely that both assumptions will hold.

THE LEARNING ALGORITHM

The question remains: how do we use the data available to us to train a model? For simplicity, we'll consider only parametric models (such as those in this chapter) and reserve discussion of nonparametric models to chapter 7.

During training, our ML model ingests training data encoded as features and learns how to map those features to the property we want to predict (figure 3.8). Our model is governed by a set of *weights* (or *parameters*). Weights are real numbers that express the importance of the model's inputs to its output. The model's weights govern how the model maps input features to output predictions. The value of each weight influences the model's predictions and, consequently, its performance. The model's performance evaluates the quality of its predictions. If the predictions are bad, which they likely will be at the beginning, then our model requires a mechanism to learn from experience, alter its weights, and improve its performance.

Learning algorithms enable automatic tuning of model weights based on performance. The learning process involves optimizing parametric model weights to achieve accurate predictions on both the training data and on new, unseen data. Each possible combination of weight values represents a separate hypothesis function. When we train our model, the model iteratively adjusts the values of these weights as it learns to recognize patterns in the data at different levels of abstraction.

With each iteration of training, the learning algorithm helps the model adjust its weights in the direction that improves performance on our training dataset. Each learning algorithm consists of three components:

1. A *loss function*, or error function. The loss function measures the error between the predicted output and the actual label or ground truth class. It represents a penalty for the model's prediction error on a single data instance.
2. An *optimization function* based on the loss function. *Cost functions* are a type of objective function commonly used in ML learning algorithms. The cost function measures an aggregate error term for the model's predictions across all data instances. For example, it could be the average of the loss functions across the full training set. With cost functions, lower is better.
3. A *learning routine* that guides weight adjustments to minimize the cost function (in general terms, to optimize the optimization function).

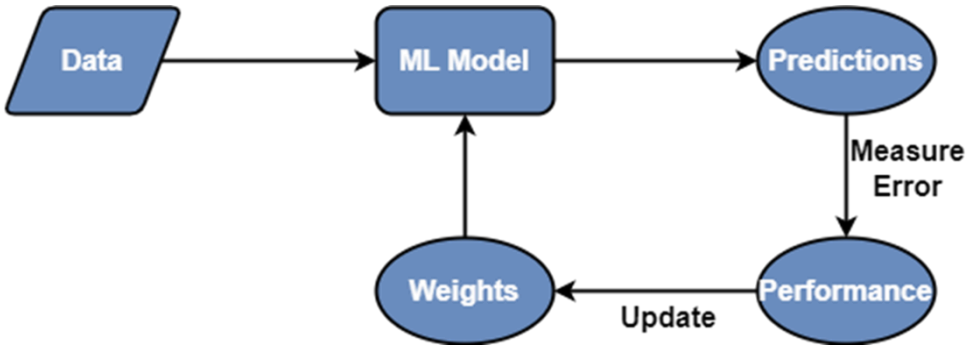

Figure 3.8 Parametric ML model during training. Our ML model learns a function that maps each input data instance to a prediction about a property of interest. The loss and optimization functions measure and aggregate the performance of the model's predictions. In tandem with a learning routine, the model's weights are updated such that, in the next iteration of training, the model will reduce its previous error and perform better.

Once we have chosen our final, trained model, we no longer incorporate weight updates. The weights are frozen and are now inherent to the model. During inference, the model simply acts on new data to generate predictions (figure 3.9).

Figure 3.9 During inference, the ML model is exposed to new, previously unseen data and outputs predictions that are used by downstream components. There is no learning as weights are *frozen* and not adjusted further.

PARAMETRIC VERSUS NONPARAMETRIC MODELS

There are many types of ML models we could choose for a given problem. Based on the assumptions made about the underlying data distribution, we can segment them into two categories:

- Parametric models make assumptions about the functional form or shape of the data distribution. These assumptions are expressed in terms of a fixed number of parameters or weights, which are learned from the training data. The concept of weights may be expressed differently in various models, but the underlying idea is to assign importance or influence to different components of the input data. For example, linear regression is a parametric model that assumes a linear relationship between the input features and the target variable.

- Nonparametric models make fewer assumptions and, instead of a fixed number of parameters, can adapt to the complexity of the data as needed. Parametric models make explicit assumptions that may not accurately represent the true data distribution, whereas non-parametric models tend to be more flexible. For example, principle component analysis (PCA) is a non-parametric technique that can simplify complex data, removing unimportant features and retaining only the most essential information without assuming the specific distribution of the data.

With a better understanding of what the model is doing with the data during training, let's train our first linear model and see how it does!

3.3.2 Training our Linear Model

Intuitively, our model acts as a function that maps molecule structure, encoded as fingerprints, to a property of interest. We'll train a linear classifier to learn the correspondence between fingerprints and ability to block the hERG channel. Figure 3.10 generalizes the process to any ML classifier and figure 3.11 depicts a simplified form of the output.

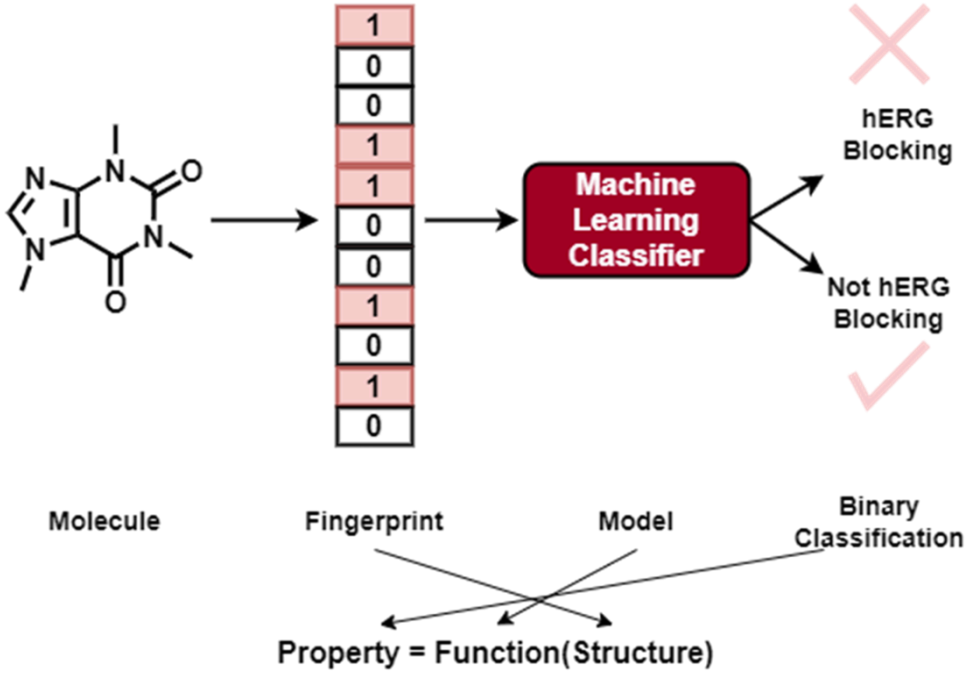

Figure 3.10 We can view our ML model as a function that learns correspondence between the structures in the input data and the output property we are modeling. To mediate this, we convert the molecule into fingerprint features that quantify its structure.

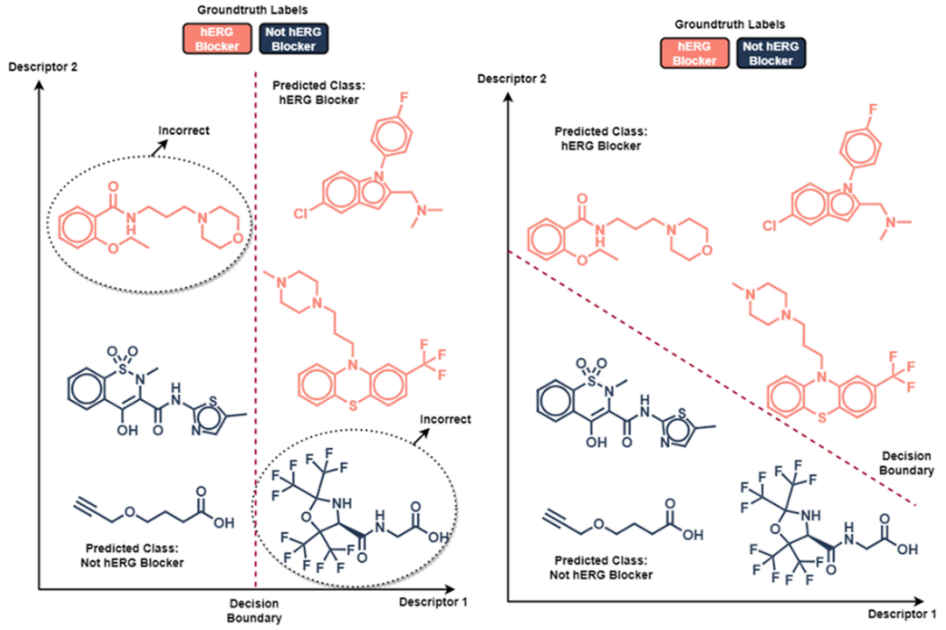

Figure 3.11 (Left) Initially, our untrained model misclassifies multiple instances. The model's decision boundary will adjust with each training iteration. (Right) Our ML model can learn a decision boundary that partitions the input data into "hERG blocker" and "not hERG blocker" classes. Assuming our data is linearly separable with respect to hERG blocker classes, we hope to develop a linear model whose decision boundary correctly separates the training data and generalizes to new data that wasn't in the training set.

The hERG blockers dataset has a pre-defined random split that partitions 392 instances into a training set and 195 instances into a test set. After we split the data, we can use the training data set to train a classifier, with many approaches available to frame this problem (see Scikit-Learn's documentation on linear models for more ideas [3]). We'll demonstrate the use of Scikit-Learn's SGDClassifier to tackle this problem. Later, in chapters 3 and 4, we'll dive into the internals of SGDClassifier to understand the underlying ML algorithms it supports and how it optimizes them.

Listing 3.7 Training a Linear Classifier

```
from sklearn.linear_model import SGDClassifier

def split_data(df, split_col="Random Split"):
  train_indices = df.index[df[split_col].str.contains("Train")].tolist()
  test_indices = df.index[df[split_col].str.contains("Test")].tolist()
  return (
    df.iloc[train_indices].sample(frac=1).reset_index(drop=True),
    df.iloc[test_indices].sample(frac=1).reset_index(drop=True)
  )

train_set, test_set = split_data(herg_blockers)
train_fingerprints = np.stack([compute_fingerprint(mol, 2, 2048) for mol in
train_set.mol])
train_labels = train_set.Class

lin_cls = SGDClassifier()
lin_cls.fit(train_fingerprints, train_labels)
```

Great, now we have a trained linear classifier. We can use it to generate predictions and also assess how good the model performs. The simplest metric for evaluating binary classifier performance is accuracy. Let's see how well our model fits to the training set.

Listing 3.8 Evaluating model performance on the training set

```
>>> from sklearn.metrics import accuracy_score
>>> herg_blockers_predictions = lin_cls.predict(train_fingerprints)
>>>herg_blockers_predictions[:10]
array([1., 0., 1., 1., 0., 1., 1., 1., 1., 1.])

>>>train_labels.iloc[:10].values
array([1., 0., 1., 1., 0., 1., 1., 1., 1., 1.])

>>> accuracy_score(train_labels, herg_blockers_predictions)
0. 9617346938775511
```

Wow! Our linear classifier is correct 96.2% of the time! Did we just solve the hERG blocker prediction challenge? Not quite. To understand why, we need to consider a few important principles related to how we evaluate our model.

3.3.3 Evaluating our Model

For us to judge our model as successful, it needs to perform well not only on data within the training set, but also generalize to unseen data. We can use the test set to assess generalization errors. However, we want to save the test set for as long as possible. If we evaluate the model on the test set and need to reiterate on the model, we will no longer have a test set! Each use of the test set contaminates it, and it becomes less and less of a good approximation of generalization error on unseen data – we want to keep from reusing it until we have a final model that we are confident in launching.

CROSS VALIDATION

We don't want to contaminate our test set. Instead, we can split our training set into a smaller training set and a validation set. We then train models on the smaller training set and compare them based on their performance on the validation set. However, we face conflicting dilemmas:

1. We want as large of a training set as possible. If there is not enough training data for our models to learn from, they won't generalize well.
2. We want as large of a validation set as possible. If our validation set is too small, it can't serve as a good approximation of how well our model's might generalize to the test set.

Ideally, we could use all instances from the original training set for model training and model validation. This would also reduce the randomness involved in picking a particular partitioning of instances into single train and validation sets. One paradigm that allows for this is *k-fold cross validation* (figure 3.12). We split the training set into *k* smaller sets, called *folds.* We then train *k* different models. Each model is trained using $\mathrm{k}-1$ of the training data folds. Each model is then validated on the remaining fold which is not used during training.

Final performance is reported as the average performance of the models on each of their validation folds. The higher value of *k*, the greater number of folds and better our approximation of generalization error at the expense of increased computation. In the extreme, leave-one-out CV sets *k* equal to the number of training instances, with one instance per fold. After determining the model with best cross-validated performance, we train the model on the full training set and proceed to final evaluation on the test set. Before deploying the model to a production environment, we would also retrain it on all available data, including the test set. Evaluating our model with 5-fold cross validation, we notice a large drop off between training accuracy and validation accuracy.

Listing 3.9 *5*-fold cross validation

```
>>> from sklearn.model_selection import cross_validate

>>> scoring = {'acc': 'accuracy'}
>>> lin_cls_scores = cross_validate(lin_cls, train_fingerprints, train_labels,
scoring=scoring, cv=5)
>>>lin_cls_scores["test_acc"].mean()
0.7755598831548199
```