Machine Learning Methods in Chemistry

James Bonnar

email: bonnarj@gmail.com

Φ

TREASURE TROVE OF MATHEMATICS

July 2021

Everything is theoretically impossible, until it is done. One could write a history of science in reverse by assembling the solemn pronouncements of highest authority about what could not be done and could never happen.

Robert A. Heinlein, 1952

Preface

In this book we discuss the technical and non-technical reasons science has been unable to find cures for heritable diseases, despite the exponential increase in knowledge of disease mechanisms we currently witness. New directions in scientific research and protocols are suggested that may help bring about actual cures for genetic diseases through pharmacological gene therapy. A computational paradigm, called the omega algorithm, is developed, implemented and applied to find compounds that could potentially correct the ΔF508 mutation responsible for cystic fibrosis. Links to downloadable files, including an extensive chemical reaction database, are given in Appendix B to assist the reader with further studies.

The chapters that follow are the first published report on the initial results of a long-term project originally conceived over fifteen years ago. At that time, I was a student of chemistry and physics at the University of Wisconsin-Parkside, near the completion of my degree. An involvement with an independent study research course in the physics department dealing with the divergences in how Bohr's correspondence principle predicts highly-energized (Rydberg) atoms should behave and their actual chaotic behavior (quantum chaos) provided my first encounter with the unknown and the insufficiency of fundamental theory to efficiently model complex systems. To my surprise, that first experience with ineffective or incomplete scientific theory and practice radically undermined most of my basic conceptions about the completeness, capabilities, validity and practice of contemporary science and the reasons for both its successes and failures.

Those conceptions were practical – I shared the same perspective on science that an engineer probably would. In this, I mean I took a constructive utilitarian point of view, rather than an analytical point of view on the ultimate objectives of science. The goal of any scientific endeavor, for me, was to ultimately be able to control, alter or build with the object of study. This instinct has deep roots in American culture and comes quite naturally to a creature possessing an opposable thumb. The results of this value system was a gradual change in my career plans over the course of my education – from premed to biochemistry to mathematics and physics. In some part the work presented in this book is a reintegration of everything I learned.

My first opportunity to vigorously pursue the ideas set forth in this book came with the development of Mathematica 7 (Stephan Wolfram et al., Wolfram Research, Inc.), but were not successful until the advent of the machine learning algorithms present in Mathematica 10. Many wrong avenues were taken along the way. Without the freedom from low-level programming tasks that Mathematica provides, the development of this new technique would have been far more difficult and most likely would not have been achieved. Much of my time in previous years was spent on the study of programming proper. In particular, I continued to program in C and Java, quite laboriously reproducing the algorithms I wished to use to answer a single question, if in fact that question could even be answered using that algorithm in practice. However, with the addition of Mathematica to my repertoire, my research has progressed at a rate 100-fold quicker. Stephan Wolfram, the original creator of Mathematica, who graduated from Caltech with his doctorate in theoretical physics at the age of nineteen, is the greatest single contributor to modern computer mathematics.

Far more than other recent scholars, Wolfram and his group has shown that a bold reconsideration of the primitives of science can be quite beneficial, though we now live in a period of rigid thought in the theoretical sciences, despite the fact that technology has become very progressive. The theoretical sciences have unfortunately become mired with politics, elitism and endeavors totally unsupported by experimental evidence.

To recap, much of my time in previous years was spent exploring fields without apparent relation to biomedical science, but in which the totality of research could be unified and generalized – the importance of which history is now bringing to light. It has become deeply set into the social order of science *not* to do this. Only specialization is rewarded or respected, and being a generalist can be misconstrued as having a lack of direction. The adage “no good deed goes unpunished” applies in the scientific world.

Fortunately, the ideas I thereby assimilated oriented my research and provided a scaffold for most of my more advanced thinking. The same orientation and scaffold gave a unity and direction to my thoughts in all of my research. Even further, my work is a direct expression of my subconscious machinery at work. Quite literally, some of my ideas and solutions came to me “in my dreams”. These forces always play a role in truly creative scientific research. Others are a testament to the way in which new experimental or computational technology may help a researcher overcome an incompatible theory. Experimental technology has a long history of inducing the formation of new theories. New computational technology will do the same in the sense that it allows scientists to seriously entertain more complex theories without the subconscious fear of not being able to do anything with the theory. In this way, my work chronicles the emergence of a new theoretical framework.

An early solidifying experience in the development of my career was the experience of being berated by a mathematician for not pursuing the mathematical approach to science very early in my academic career. The experience made a lasting impression on me. There exists very different attitudes about how science should be done in the non-mathematical versus the mathematical sciences (by non-mathematical sciences I mean biology and most of chemistry, and by mathematical sciences I mean physics, engineering, computer science and mathematics itself). The number and extent of disagreements between these two groups concerning the nature of true science and how it should be done is surprising. But history forces me to doubt that the mathematics-based natural sciences are any more legitimate or permanent in their conclusions than the non-mathematical sciences are. It is not a good thing to get overly impressed by the existence of a mathematical model to describe a theory. It does not necessarily impart truth to the theory. Mathematics is infinitely flexible – it can model anything, whether we assign true or false meaning to the equations is a matter of interpretation.

Yet, somehow, the practice of physics and engineering fail to evoke the same frustration over fundamental objectives that are endemic among fields such as cancer research or chemical synthesis. Meditating upon the source of that difference in these two communities led me to the realization that learned roles play a dominant part in scientific research. These are universally-recognized (and enforced) modes of operation that provide not only a subset of admissible problem domains to a group of practitioners, but also a limited subset of admissible *solution domains* to that group. Biologists don't typically scribble partial differential equations on the chalkboard when discussing gene expression, nor do chemists talk about the latest algorithmic advancements in numerical analysis when discussing molecular dynamics calculations. Molecular biologists attempt gene therapy with molecular biology tools (virus vectors). Chemists treat disease with small organic molecules relatively easy to synthesize. These seem like reasonable modes of operation only because cultural expectation allows for them and traditions demand them. Once this realization occurred, my research direction was legitimized and justified in my mind, and a new outlook emerged.

Since my most important objective is to change the way familiar systems are evaluated, the occassional sketchiness in this book is no drawback. I want the readers to use their imaginations. Chance favors the prepared mind, as the saying goes. If the reader's own frame of mind is open to the sort of suggestions given, he or she may find the material much easier to learn and digest, and improve upon.

The take on science developed in my research suggests several new avenues of investigation which I'm convinced will prove fruitful. And the manner in which unexpected results occurred has gained my attention – each of these results merits further detailed study. In my view, every scientific discovery worth publishing alters the perspective of the person reading about it. Then that change of perspective itself should have an effect upon the content of future publications and research.

CONTENTS

1

A New Light

What I would like to do in this chapter is elucidate a bit the scientific structure I spent the last twenty years struggling to build – a structure which in the end proved embarrassingly simple to implement. It is a way of thinking which has some very exciting implications both now and well into the future. I had gotten interested in the semantics of representations (assignment of meaning to syntax) and had also been thinking about the equivalence of different representations in physics. For example, the equivalence of matrix and wave mechanics. Equally important, I had also been studying the correspondence principle and had been meditating a great deal on the implications of this phenomena in terms of computational efficiency. Newtonian calculations on large scale objects are far more efficient than sum total quantum mechanical calculations would be on the object's parts.

Even orthodox viewpoints often demonstrate the fact that simplicity can be found in the most complex phenomena of nature. What is unorthodox about my viewpoint is the belief that simplicity can *always* be found in the complex, if one is willing to see things in a new light. From the high-level differential equations that describe turbulent fluid flows, to the wave mechanics describing electron behavior about the nucleus, these are all examples of simplicity found in phenomena which could be treated at far more complicated deeper levels. Deepening a theory brings increased understanding, but not necessarily an ability to compute things better. There is actually merit in looking for what one may think of as more superficial theories for the purposes of doing computations efficiently. Two theories which say very different things in the general may say essentially the same thing, they are equivalent, in some limited domain. I believe I have found a way to derive computationally-efficient, secondary theories to model phenomena already explained by computationally-inefficient primary theories within certain limited problem domains. Specifically, we shall be extracting the knowledge inherent in a chemical reaction database and applying it to make predictions about reactions involving complex molecular species, namely, determination of molecules to react with a ssDNA sequence to transform the sequence in some desired way – this process is called the omega algorithm. What we will compute is in effect a necessary but not-necessarily-sufficient condition that a proposed molecule would produce the transformed sequence when reacted with the initial sequence of DNA. This allows us to tremendously narrow down our search of chemical space for the proper reactant in an effort to cure a heritable disease.

It is a common misconception that the study of complex, deep phenomena such as say, quantum field theory, will enable one to easily figure out what is going on in higher-level systems such as say, a methane molecule. It seems reasonable – the whole is just the sum of its parts, right? But when one tries to actually do such a thing, things rarely work out, except in the simplest of cases. Reductionism has its benefits, but putting something back together is not the mirror inverse of tearing it apart, especially in regards to computation on complex microscopic systems such as biomolecules. Gradually, I began to realize that there is a fundamental problem with the whole approach science is using to describe such systems. It is commonly believed that to describe such systems, one must build from the bottom up. But this is a fallacy. In the first place, we are not truly building from the bottom up (how many chemists use string theory to describe the hydrogen molecule?). Secondly, I have found experimentally that it is more productive, computationally-efficient theories can always be found, if one uses *surface-level theories*. By surface-level theory, I mean not a lower-level or higher-level theory, but a theory that employs data concerning the level about which the theory is meant to make predictions. This in no way minimizes the importance of microscopic descriptions of matter, but what I mean by this is that it is more effective to use molecular structure itself as the data from which we build our theories about the behavior of molecular structures. It is *not* necessary to build up from say, mathematical models intended to describe the electrons within atoms, to describe say, a protein's catalytic behavior (at least as far as computing its behavior).

But one may be asking how we are to find such theories. The quantum theory of molecular structure itself was so hard won. How are we to find these new surface-level theories? The good news is that it is not only possible, but the process of finding these mathematical models has been automated and are very easily implemented. Computationally-efficient theories of chemical reactivity can and have been found using nearest-neighbors analysis of a database of SMILES strings that describe chemical reactions, but we will discuss more on that later.

If one looks at history, the use of mathematics to describe natural phenomena has been a defining feature of the advanced sciences. It has been observed that the more advanced the science, the more mathematical it becomes. Even in biology, we now see mathematics making an inroad in the form of data analysis or what has come to be called "bioinformatics". But despite the existence of subfields such as "chemical physics", one may be surprised to learn that such subfields often have little to do with the practice of the rest of the field. For example, organic chemists deal with synthesizing molecules of such high complexity that physical chemists have little chance to add anything helpful to the organic chemist's work in practice. The theoretical does provide understanding, but is often of little to no predictive value because the computations cannot be done.

Imagine an alternate universe in which the equations of electromagnetism were so inscrutable to numerical computation that electrical engineers could not do reliable computations on systems any more complex than the common light bulb. In this world, engineers could analyze and understand even the most complex supercomputers found laying around, but could not build a radio if their lives depended on it. At the finest universities, courses are taught on subjects such as "combinatorial electronics", in which students are taught how to build random appliances and develop assays to detect good appliances. These are precisely the strange set of circumstances under which chemists must live. They must do so because the equations they have been taught *must* be used to do legitimate, acceptable computations on chemical systems are not amenable to accurate numerical computation in regards to complex systems. This is a result of conditioning and social pressure, not scientific necessity.

Is there someway to go beyond the accepted paradigm in thinking about complex systems? What are the necessary ingredients of a computation concerning a system we know is composed of "parts"? How is it that nature can form complex systems with equal ease as it forms simple systems? In looking for the ingredients or "primitives" of an efficient computation, might we also be inadvertently also be learning something about nature itself? It was discovered many years ago by Richard Feynman that the movement of subatomic particles could be accounted for if, in moving from place to place, the particle was considered to take *all* possible paths. These paths add up, but only some of them add to the sum significantly. So nature itself, even at its deepest levels, displays this tendency of computational efficiency. The crucial thing to understand is that these primitives do not need to be based on the smallest parts of the complex system.

But if one is to do computations on natural systems at all, then one is going to have to implicitly assume that nature follows definite rules, which appears to be the case. But exactly who is to define what the *complete* set of rules are for us? In nature, there are rules, but then there are rules governing the rules, and then there are rules governing the rules that govern the rules, and so forth. The classical laws of physics must be satisfied by quantum mechanics in the limit of large numbers, the laws of ordinary quantum mechanics must be satisfied by string theory, and so on. Is there any way to make these observations systematic and useful? In the past, there was no way to do so, but now we have computers. What sort of computer program is relevant to the determination of computational primitives? In programming we are used to developing long, complicated programs suited to a particular task. But here we need a way to derive, in general, the efficient computational primitives of a problem domain, in particular those of chemical reactivity. Such an algorithm was implemented by the author and is presented in this book – it is a machine learning algorithm, a form of *string transformation learning*, for equating ordered pairs of character strings to a single number, deriving just what the rule or map is governing the relation between the ordered pairs in general, and then using this rule to make predictions on novel cases in the domain of the map. In this scheme the strings of characters represent molecules. I have coined the algorithm for finding reactants that transform an initial strand of DNA to some desired product strand of DNA the *omega algorithm.*

One will see that here we have a systematic method for finding the computational primitives for complex molecular systems and this provides a way of doing all the computer experiments we shall have to do in an efficient way. For many years, developing a method for finding these computational primitives became an obsession. As Linus Pauling would say, the best way to get good ideas is to get a lot of ideas. Finally, the method has developed to the point where I can start looking at the questions in science, particularly in medicinal chemistry or gene therapy, that I want to. I feel very much like I am using a tool that can be simply pointed at a problem and immediately see the phenomenon in a new light, a computationally-efficient one at that.

Very simple rules can give incredibly complicated, even surprising behavior. It is a robust, general phenomenon. How come a more fundamental approach such as this hadn't been in use in the molecular sciences for decades? One reason is, until recently, high-speed computers were not widely available. One can see these things only by doing many computer experiments. With our ordinary intuition alone, there just did not seem to be any reason to even try these computer experiments. It seemed so obvious that they wouldn't show anything interesting, at the time. We were wrong. But now, we can see hints of this phenomenon from the past, even in mathematics itself. As an example, the sequence of primes, irregular and able to pass any test of randomness we have, can be generated from a simple deterministic formula. Similarly, the digits of π can also be generated from a deterministic formula, though they can also pass any test of randomness. In the same way, nature itself seems to produce so much that seems, to us, so complicated. But complexity is really only a matter of perspective. It is as if nature has some secret for building complex systems.

Many scientists believe that in order to create things we have to operate under the constraint that we have to foresee what the things we create are going to do. So many have forced themselves into a limiting solution space of special algorithms which only have predictable, foreseeable behavior. Presumably, nature is under no such constraint, in that way somehow producing the degree of complexity we humans fail to emulate in our creations. Using the protocols of string transformation learning, it may now be possible to synthesize molecules and predict reactions of very complex molecules. As Richard Feynman wrote, “What I cannot create, I do not understand.”, which was written on his chalkboard at his time of death in 1988.

Some scientists may feel uncomfortable with using these new surface-level models when they tackle a problem. Some have a tendency to seek the approval of others in how they go about solving a problem. This is where orthodoxy comes from. But the whole point of *any* model is to capture certain *essential* features of a system and to idealize away everything else. If the objective is computation, what could the essential features be, other than the computational primitives needed to perform efficiently? Depending on what aspect one is interested in, one selects certain features to capture. A common misconception about models is that the models are supposed to "be" the system itself. A model is an abstract way of reproducing what a system *does*. Its the same with this string transformation learning technique – the system represents abstractly *what happens* when molecular species encounter one another. What type of model is best will depend on what aspect of molecular behavior one is interested in. We do not have a single model to account for everything, just a single protocol for arriving at the models.

2

History and Epistemology

“The fundamental laws necessary for the mathematical treatment of a large part of physics and the whole of chemistry are thus completely known, and the difficulty lies only in the fact that application of these laws leads to equations that are too complex to be solved.” – Paul Dirac, 1929.

This well-meaning, but unfortunate, statement of Dirac’s clearly expresses the belief held by most scientists to this day that computational primitives equate with the fundamental parts of a system. If this statement could be erased from history and rewritten, I would write it so:

“The fundamental laws necessary for the mathematical understanding of a large part of physics and the whole of chemistry are thus completely known, and the difficulty lies only in the fact that direct application of these laws leads to equations that are too complex to be solved.”

There is an old adage that claims that a whole is *greater* than the sum of its parts. What that actually means is that a whole is *different* than a straightforward sum of its parts. But one finds in reality, computationally speaking, that wholes are often *less* than the sum of their parts. The Earth travels about the Sun in a fairly simple trajectory determined by a simple equation. We do not have to solve for the quantum mechanical orbits of each subatomic particle comprising the system.

I ask the question, is it actually necessary to use very detailed quantum mechanical calculations to predict the outcome of a chemical reaction? In light of what we were taught, the way we were trained to think, the question seems almost ridiculous to pose, but is it? Considering the human cost of the *limitations* of current theoretical chemistry, it seems worth pondering. Just as an astrophysicist would not calculate the orbit of the Moon around the Earth by taking into account the quantum mechanical orbits of each subatomic particle composing it, to what degree can chemists do the same? Seems we should not be able to get very far – "the molecules are more proximal to the quantum mechanical scale and thus ought to be more tightly ruled by it", one might say. However that is false reasoning because the Moon *is* just as tightly ruled by quantum mechanics as are electrons, so says the correspondence principle. Quantum mechanics is correct at any length scale we have been able to probe, classical physics being more an approximate statement of the properties of quantum mechanics in the limit of large numbers. Classical physics is an *epiphenomenon* – a secondary phenomenon that occurs alongside or in parallel to a primary phenomenon – quantum physics, in this case. Thus, we cannot conclude that simplifying epiphenomena (side effects) do not exist for molecular objects as well – epiphenomena unrelated to scale, but more related to the attainment of stable intermolecular states. Further, when I was an organic chemistry student many years ago, I was at that point in time not knowledgeable in the field of quantum mechanics, yet I was able to predict the outcomes of thousands of organic reactions. How was I able to learn to do that? Thus, the epiphenomena, the rules of chemistry, must exist, it's just that we humans are having difficulty articulating them in an explicit way. String transformation learning solves that problem.

I want to state a principle that I am certain is universally true – The Epiphenomenological Principle: All physical laws are secondary. Physical laws always change as we drill down into the infinitesimal without end, and out into infinity without end. Physical laws, in reality, are no more "real" than the laws describing the behavior of an ant colony or school of fish. Mathematical laws, on the other hand, are not epiphenomenological. The moral I am trying to relate here is that it is not at all inappropriate to look for ways to do chemistry outside of strict quantum mechanical approaches, which have serious limitations that affect human welfare. I am *not* saying quantum mechanics isn't the best way to appraoch *some* problems. I don't want to be black-and-white about things here. But let's not exclude other facets of nature which could be very rewarding.

A theory that is a bit further away from reality than a more accurate theory, however is superior in its imagery, symbolism and its ability to rationalize scientific fact may retain its value in the community. For example, consider the famous rivalry between valence bond theory and molecular orbital theory. The two theories were developed at about the same time, but quickly diverged into rival schools that have competed, sometimes fervently, on charting the mental map and epistemology (knowledge) of chemistry. It has been argued over the years that valence bond (VB) theory is flawed, and that molecular orbital (MO) theory is superior for explaining things like the structure of benzene. However, VB theory is central to the chemist's basic concept of what a chemical bond is. The once held belief that theories should automatically be thrown away in situations such as this is incognizant of the fact that even MO theory is a bit far from reality itself, and could be replaced with a quantum electrodynamic theory of the molecule, and so on. Each theory must be valued for the unique merits it has, assuming it has substantial unique merits, as VB theory does. There exist acceptable levels of mythhology in science, and I would even go so far as to say that often, a sufficient degree of mythology is preferable. Consider the case of classical mechanics – a false, but delightful approximation. So I must ask, is any science permanent? Phrased this way, perhaps. But never permanently considered "ultimate truth".

In light of the above, some rules of effective research can be gleaned from the history of science that better describe the actual mode of operation of great scientists than the so-called "scientific method" does.

Four Questions Effective Research Answers
What are the fundamental entities of which problem space is composed?
How do the fundamental entities interact?
What techniques are employed in defining the problem space and how do we justify the interpretation of experimental data?
How is the solution space best defined?

A common misconception about science is that the invention of new theory is the most fundamental act in science. This is not true. In fact, the most fundamental act is reinterpretation of experimental facts. By "theoretical" science, it is clear that the theorizing that is being done is concerned with the interpretations of data. We theorize about X-rays, hydrogen atoms, etc. But the more fundamental act is the assignment of causation to, or even further, the existence of, any such object.

Each scientific field has an admissible problem space (understandably), and an admissible solution space (unfortunately). Of the physical sciences, physics seems to have the most complete solution space, borrowing from mathematics to the point of defining physics as the mathematical science. The connection of chemistry and biology to mathematics is once and twice removed. Biology borrows heavily from chemistry, which in turn borrows from physics. There is very little direct interaction between mathematics and chemistry or biology. I find this situation unacceptable. There should be a "chemical mathematics" as well as a "biological mathematics". These fields need a solution space of their own, defined in terms of mathematics in a direct fashion. So far, the greatest achievements in the sciences have been attained when one field borrows and applies advancements made in another field. In the molecular sciences, this has meant the application of physics. But it has become clear that a direct route must be laid between mathematics and chemistry in particular.

New theories do not emerge from old theories (and the belief that we must use complex quantum mechanical codes to calculate chemical reaction outcomes is a theory). New theories require the reevaluation of old theories. It is the old theories which must be readjusted to the new theories.

3

Software You'll Need

There are two sets of programs you will need to execute the code in this book, Mathematica and Marvin Suite.

3.1 Mathematica

Wolfram Research's https://www.wolfram.com/mathematica/ Mathematica is a symbolic mathematical computation program, sometimes called a computer algebra program, used in many scientific, engineering, mathematical and computing fields. It was conceived by Stephan Wolfram and is developed by Wolfram Research of Champaign, Illinois. The Wolfram Language is the programming language used in Mathematica. **To execute the code in this book, you'll need Mathematica version 10 or later.**

3.2 Marvin Suite

ChemAxon https://www.chemaxon.com allows for free download of the Marvin Suite of products, which contains MarvinSketch, MarvinView, MarvinSpace and other programs. In particular, one will be needing MarvinSketch to draw in molecules and to save them as SMILES strings, and MarvinView to view lists of SMILES strings as 2-dimensional structural diagrams.

4

SMILES Strings

The omega algorithm, a machine learning algorithm, requires that we form an abstract model of chemical reactivity data and we then use that model to make predictions about the reactivity of novel compounds. The most important issue in any application of machine learning is the representation of information that is fed into the system and/or obtained from it. The representation chosen has to be adapted to the problem and solution space. We need a way to represent molecular structures that captures all of the structural facets of their connectivity, yet consists of linear strings of characters such that Mathematica's machine learning function `Predict` can work with the representation. The requirements of our representation are satisfied by the SMILES system of specifying molecular structure. As we will see in the next chapter, the SMILES representation is also ideal for automatically generating novel molecules.

The *simplified molecular-input line-entry system* (SMILES) is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings. SMILES strings can be imported by most molecule editors for conversion back into two-dimensional drawings or three-dimensional models of the molecules.

The original SMILES specification was initiated by David Weininger at the USEPA Mid-Continent Ecology Division Laboratory in Duluth in the 1980s. Acknowledged for their parts in the early development were Gilman Veith and Rose Russo (USEPA) and Albert Leo and Corwin Hansh (Pomona College) for supporting the work, and Arthur Weininger (Pomona, Daylight

CIS) and Jeremy Scofield (Cedar River Software, Renton, WA) for assistance in programming the system. The Environmental Protection Agency funded the initial project to develop SMILES.

SMILES contains the same information as might be found in an extended connection table. The primary reason SMILES is more useful than a connection table is that it is a linguistic construct, rather than a computer data structure. SMILES is a true language, albeit with a simple vocabulary and only a few grammar rules. SMILES representations of structure can in turn be used as “words” in the vocabulary of other languages designed for storage of chemical information and chemical intelligence.

Part of the power of SMILES is that unique SMILES exist. With standard SMILES, the name of a molecule is synonymous with its structure; with unique SMILES, the name is universal. Anyone in the world who uses unique SMILES to name a molecule will chose the exact same name.

4.1 Canonicalization

SMILES denotes a molecular structure as a graph with optional chiral indications. This is essentially the two-dimensional picture chemists draw to describe a molecule. SMILES describing only the labeled molecular graph (i.e., atoms and bonds, but no chiral or isotopic information) are known as generic SMILES. There are usually a large number of valid generic SMILES which represent a given structure. A canonicalization algorithm exists to generate one special generic SMILES among all valid possibilities; this special one is known as the “unique SMILES”. SMILES written with isotopic and chiral specifications are collectively known as “isomeric SMILES”. A unique isomeric SMILES is known as an “absolute SMILES”. The following table gives some examples:

Input SMILES	**Unique SMILES**
`OCC`	`CCO`
`[CH3][CH2][OH]`	`CCO`
`C-C-O`	`CCO`
`C(O)C`	`CCO`
`OC(=O)C(Br)(Cl)N`	`NC(Cl)(Br)C(=O)O`
`ClC(Br)(N)C(=O)O`	`NC(Cl)(Br)C(=O)O`

4.2 SMILES Specification Rules

SMILES notation consists of a series of characters containing no spaces. Hydrogen atoms may be omitted (hydrogen-supressed graphs) or included (hydrogen-complete graphs). Aromatic structures may be specified directly or in Kekulé form.

There are five generic SMILES encoding rules, corresponding to specification of atoms, bonds, branches, ring closures, and disconnections. Rules for specifying various kinds of isomerism also exist.

Atoms

Atoms are represented by their atomic symbols: this is the only required use of letters in SMILES. Each non-hydrogen atom is specified independently by its atomic symbol enclosed in square brackets, `[ ]`. The second letter of two-character symbols must be entered in lower case. Elements in the "organic subset" `B`, `C`, `N`, `O`, `P`, `S`, `F`, `Cl`, `Br`, and `I` may be written without brackets if the number of attached hydrogens conforms to the lowest normal valence consistent with explicit bonds. "Lowest normal valences" are `B`(3), `C`(4), `N`(3,5), `O`(2), `P`(3,5), `S`(2,4,6), and 1 for the halogens. Atoms in aromatic rings are specified by lower case letters, e.g., aliphatic carbon is represented by the capital letter `C`, aromatic carbon by lower case `c`. Since attached hydrogens are implied in the absence of brackets, the following atomic symbols are valid SMILES notations:

`C`	methane	CH4
`P`	phosphine	PH3
`N`	ammonia	NH3
`S`	hydrogen sulfide	H2S
`O`	water	H2O
`Cl`	hydrochloric acid	HCl

Atoms with valences other than "normal" and elements not in the "organic subset" must be described in brackets.

`[S]`	elemental sulfur
`[Au]`	elemental gold

Within brackets, any attached hydrogens and formal charges must always be specified. The number of attached hydrogens is shown by the symbol `H` followed by an optional digit. Similarly, a formal charge is shown by one of the symbols `+` or `-`, followed by an optional digit. If unspecified, the number of attached hydrogens and charge are assumed to be zero for an atom inside brackets. Constructions of the form `[Fe+++]` are synonymous with the form `[Fe+3]`. Examples are:

[H+]	proton
[Fe+2]	iron (II) cation
[Fe++]	iron (II) cation
[OH-]	hydroxyl anion
[OH3+]	hydronium cation
[NH4+]	ammonium cation

Bonds

Single, double, triple and aromatic bonds are represented by the symbols `-`, `=`, `#` and `:`, respectively. Adjacent atoms are assumed to be connected to each other by a single or aromatic bond (atoms involved in aromatic bonds being distinguished by the fact that can be designated by lower case letters). Single and aromatic bonds may always be omitted. Examples are:

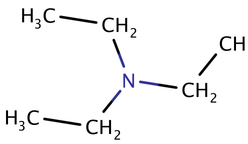
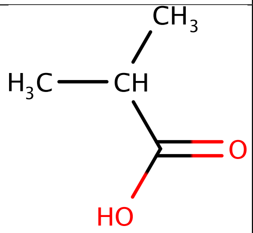
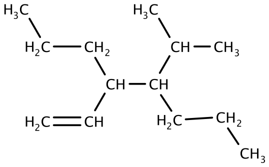
CC	ethane	CH3CH3
C=O	formaldehyde	CH2O
C=C	ethene	CH2=CH2
O=C=O	carbon dioxide	CO2
COC	dimethyl ether	CH3OCH3
C#N	hydrogen cyanide	HCN
CCO	ethanol	CH3CH2OH
[H][H]	molecular hydrogen	H2

For linear structure, SMILES notation corresponds to conventional diagrammatic notation except that hydrogens and single bonds are generally omitted. For example, 6-hydroxy-1,4-hexadiene can be represented by many equally valid SMILES, including the following three:

Structure	Valid SMILES
	C=CCC=CCO
CH2=CH-CH2-CH=CH=CH-CH2-OH	C=C-C-C=C-C-O
	OCC=CCC=C

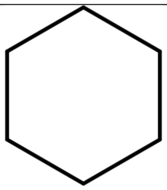
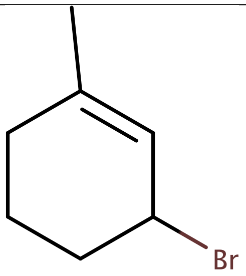
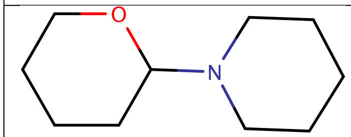
Branches

Branches are specified by enclosing them in parentheses, and can be nested or stacked. In all cases, the implicit connection to a parenthesized expression is to the left. Examples are:

Triethylamine	**Isobutyric acid**	**3-propyl-4-isopropyl-1-heptene**
`CCN(CC)CC`	`CC(C)C(=O)O`	`C=CC(CCC)C(C(C)C)CCC`
 H_3C CH_2 N CH_2 CH_3 H_3C CH_2		 H_3C H_2C CH_2 H_3C CH CH_3 CH CH H_2C CH H_2C CH_2 CH_3

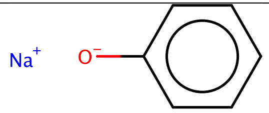
Cyclic Structures

Cyclic structures are represented by breaking one bond in each ring. The bonds are numbered in any order, designating ring opening (or ring closure) bonds by a digit immediately following the atomic symbol at each ring closure. Cyclohexane is an easy example.

	`C1CCCCC1`
	`CC1=CC(Br)CCC1`
	`O1CCCCC1N1CCCCC1`

Disconnected Structures

Disconnected compounds are written as individual structures separated by a "." (period). The order in which ions or ligands are listed is arbitrary. There is no implied pairing of one charge with another, nor is it necessary to have a zero net charge. If desired, the SMILES of one ion may be imbedded within another as shown in the example of sodium phenoxide:

	`[Na+].[O-]c1ccccc1`

4.3 Isomeric SMILES

This section describes the SMILES rules used to specify isotopism, configuration about double bonds, and chirality. The term *isomeric SMILES* collectively refers to SMILES written using these rules.

The SMILES isomer specification rules allow chirality to be completely specified for any structure, if it is known. Unlike most existing chemical nomenclatures such as CIP and IUPAC, these rules are also designed to allow rigorous partial specification of chirality. Aside from use in macros, substructure searching, and other pattern matching operations, this is important because much of the world's available chemical information is known for structures with incompletely resolved chiralities (not all possible chiral centers are separated, known, or reported).

All isomer specification rules in SMILES are therefore optional. The absence of a specification for any attribute implies that the value of that attribute is unspecified.

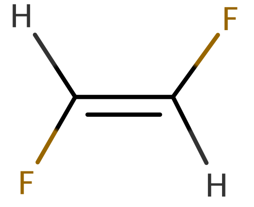
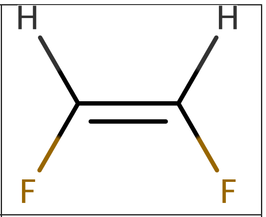
Isotopic Specification

Isotopic specifications are indicated by preceding the atomic symbol with a number equal to the desired integral atomic mass. An atomic mass can only be specified inside brackets. For instance:

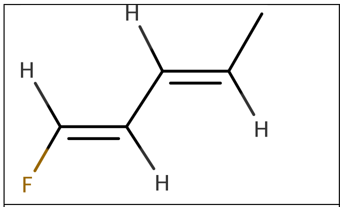
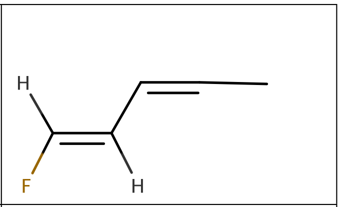
SMILES	Name
[12C]	carbon-12
[13C]	carbon-13
[C]	carbon (unspecified mass)
[13CH4]	C-13 methane

Configuration Around Double Bonds

Configuration around double bonds is specified by the characters / and \ which are "directional bonds" and can be thought of as kinds of single or aromatic bonds. These symbols indicate relative directionality between the connected atoms, and have meaning only when they occur on both atoms which are double bonded. For instance, the following SMILES are valid for E- and Z-1,2-difluoroethene:

	
F/C=C/F	F/C=C\F
F\C=C\F	F\C=C/F

An important difference between SMILES chirality conventions and others such as CIP is that SMILES uses a local chirality representation (as opposed to absolute chirality), which allows partial specifications. An example of this is illustrated here:

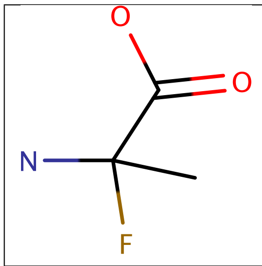
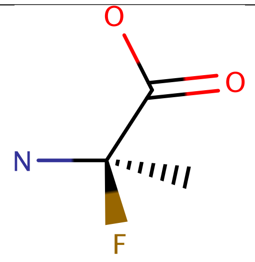
	
F/C=C/C=C/C	F/C=C/C=CC
completely specified	partially specified

Configuration Around Tetrahedral Centers

SMILES uses a very general type of chirality specification based on local chirality. Instead of using a rule-based numbering scheme to order neighbor atoms of a chiral center, orientations are based on the order in which neighbors occur in the SMILES string. As with all other aspects of SMILES, any valid order is acceptable.

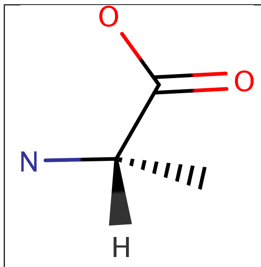
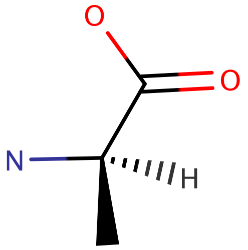
The simplest and most common kind of chirality is tetrahedral; four neighbor atoms are evenly arranged about a central atom, known as the “chiral center”. If all four neighbors are different from each other in any way, mirror images of the structure will not be identical. The two mirror images are known as “enantiomers” and are the only two forms that a tetrahedral center can have. If two (or more) of the four neighbors are identical to each other, the central atom will not be chiral (its mirror images can be superimposed in space).

In SMILES, tetrahedral centers may be indicated by a simplified chiral specification (@ or @@) written as an atomic property following the atomic symbol of the chiral atom. If a chiral specification is not present for a chiral atom, its chirality is implicitly not specified. For instance:

	
NC(C)(F)C(=O)O	N[C@](C)(F)C(=O)O
NC(F)(C)C(=O)O	N[C@@](F)(C)C(=O)O
unspecified chirality	specified chirality

Looking from the amino N to the chiral C (as the SMILES is written), the three other neighbors appear anticlockwise in the order that they are written in the top SMILES, and clockwise in the bottom one. The symbol @ indicates that the following neighbors are listed anticlockwise, whereas @@ indicates that the neighbors are listed clockwise.

If the central carbon is not the very first atom in the SMILES and has an implicit hydrogen attached (it can have at most one and still be chiral), the implicit hydrogen is taken to be the first neighbor atom of the three neighbors that follow a tetrahedral specification. If the central carbon is first in the SMILES, the implicit hydrogen is taken to be the "from" atom. Hydrogens may always be written explicitly (as `[H]`) in which case they are treated like any other atom. In each case, the implied order is exactly as written in SMILES. For example, some of the valid SMILES for alanine are:

	
`N[C@@]([H])(C)C(=O)O`	`N[C@]([H])(C)C(=O)O`
`N[C@@H](C)C(=O)O`	`N[C@H](C)C(=O)O`
`N[C@H](C(=O)O)C`	`N[C@@H](C(=O)O)C`
`[H][C@}(N)(C)C(=O)O`	`[H][C@@](N)(C)C(=O)O`
`[C@H](N)(C)C(=O)O`	`[C@@H](N)(C)C(=O)O`

There are many kinds of chirality other than tetrahedral. More advanced chiral specifications are possible with SMILES but will not be covered here.

4.4 SMILES Conventions

Aside from the above rules, a small number of conventions are universally used in SMILES. These are briefly discussed below.

Hydrogens

Hydrogen atoms do not normally need to be specified when writing SMILES for most organic structures. The presence of hydrogens may be specified in three ways:

- Implicitly ... for atoms specified without brackets, from normal valence assumptions.

- Explicitly by count ... inside brackets, by hydrogen count supplied; zero if unspecified.
- As explicit atoms ... as `[H]` atoms.

There is no distinction between "organic" and "inorganic" SMILES nomenclature. One may specify the number of attached hydrogens for any atom in any SMILES. For example, propane may be entered as `[CH3][CH2][CH3]` instead of `CCC`.

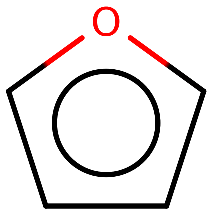
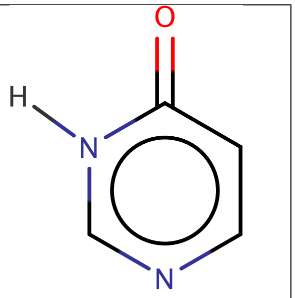
There are four situations where specification of explicit hydrogen specification is required:

- charged hydrogen, i.e., a proton, `[H+]`
- hydrogens connected to other hydrogens, e.g., molecular hydrogen, `[H][H]`
- hydrogens connected to other than one other atom, e.g., bridging hydrogens
- isotopic hydrogen specifications, e.g., heavy water, `[2H]O[2H]`

Aromaticity

Aromaticity must be deduced in a system such as SMILES which generates an unambiguous chemical nomenclature because of the fundamental requirement to characterize the symmetry of a molecule. Given effective aromaticy-detection algorithms, it is not necessary to enter any structure as aromatic if the user prefers to enter an aliphatic (Kekulé-like) structure. Entering structures as aromatic directly (i.e., by using lower case atomic symbols) provides a shortcut to accurate chemical specification and is closer to the mental molecular model used by most chemists.

The SMILES algorithm uses an extended version of Hueckel's rule to identify aromatic molecules and ions. To qualify as aromatic, all atoms in the ring must be sp^2-hybridized and the number of available "excess" p-electrons must satisfy Hueckel's 4N+2 criterion. As an example, benzene is written `c1ccccc1`, but an entry of `C1=CC=CC=C1` - cyclohexatriene, the Kekulé form - leads to detection of aromaticity and results in an internal structural conversion to aromatic representation. Conversely, entries of `c1ccc1` and c1ccccccc1 will produce the correct anti-aromatic structures for cyclobutadiene and cycleoctatetraene, C1=CC=C1 and C1=CC=CC=CC=C1. In such cases the SMILES system looks for a structure that preserves the implied sp^2 hybridization, the implied hydrogen count, and the specified formal charge, if any. Some inputs, however, may not only be incorrect but also impossible, such as c1cccc1. Here c1cccc1 cannot be converted to C1=CCC=C1 since one of the carbon atoms would be sp^3 with two attached hydrogens. In such a structure alternating single and double bond assignments cannot be made. The SMILES system will flag this as an "impossible" input. Note that only C, N, O, P, S, As, Se, * (wildcard) atoms can be considered aromatic.

	
C1=COC=C1	C1=CN=C[NH]C(=O)1
c1cocc1	c1cnc[nH]c(=O)1

4.5 Extensions for Reactions

The SMILES language is extended to handle reactions. There are two areas where SMILES is extended: distinguishing component parts of reactions and atom maps. We will not discuss atom maps here.

Component parts of a reaction are handled by introducing the > character as a new separator. Any reaction must have exactly two > characters in it. Each of the >-separated components of a reaction must be a valid SMILES.

As an aside, molecule SMILES never have a > character. In a program, one can quickly determine if a SMILES refers to a reaction or molecule by searching for a > character in the string.

Examples of reaction SMILES are:

```
C=CCBr>>C=CCl
```

This is a valid SMILES reaction. Note that there are no agent molecules, which would appear between the two `>` symbols. Also note that several atoms are missing from the reaction.

```
[I-].[Na+].C=CCBr]>>[Na+].[Br-]C=CCl
```

This is a more complete version of the same reaction. It has been canonicalized.

```
C=CCBr.[Na+].[I-]>CC(=O)C>C=CCL.[Na+].[Br-]
```

This version of the reaction includes an agent. Note that the SMILES does not indicate how the agent participates. Whether the agent is a solvent, catalyst, or performs another function within the reaction must be stored separately as data.

4.6 SMILES Chemical Reaction Database

The omega algorithm requires that we form an abstract model of chemical reactivity. We will compute a map from reactants to products. A chemical reaction database, the SMILES chemical reaction database, will be used for this purpose and the reader may download it at the address given in Appendix B. The SMILES Chemical Reaction Database has two million entries and is contained in six files, `rxnlist01.smiles` through `rxnlist06.smiles`. All of the SMILES reaction strings in the database follow the pattern `reactants>>products`, have no information regarding agents, and are separated by newline characters. They can be read into MarvinView or, one at a time, into MarvinSketch.

We will only be able to work with a small subset of the entire database. We will extract reactions that are bimolecular and involve large reactants, such that the machine learning system would learn something about the complications involved in transforming large reactants, as in a reaction involving the specific transformation of DNA. We will also concentrate on reactions involving only the organic subset of elements.

Some of these things are accomplished in the first Mathematica notebook we discuss, `rxnsfilemaker.nb`, which can be downloaded from the address in Appendix B. We shall discuss that notebook here. The first step is to set your active directory to where you have the database files stored (change the directory name to your own directory name):

```
SetDirectory["/home/james/Desktop/rxns"]
```

```
/home/james/Desktop/rxns
```

You can check your current directory as follows:

```
Directory[]
```

```
/home/james/Desktop/rxns
```

You can see the files in the current directory as follows:

```
FileNames[]
```

```
{rxnlist01.smiles, rxnlist02.smiles, rxnlist03.smiles,
 rxnlist04.smiles, rxnlist05.smiles, rxnlist06.smiles}
```

Next we import each of the SMILES database files as a `List`:

```
list1 = Import["rxnlist01.smiles", "List"];
list2 = Import["rxnlist02.smiles", "List"];
list3 = Import["rxnlist03.smiles", "List"];
list4 = Import["rxnlist04.smiles", "List"];
list5 = Import["rxnlist05.smiles", "List"];
list6 = Import["rxnlist06.smiles", "List"];
```

We join each of the lists into one large list:

```
list = list1 ~ Join ~ list2 ~ Join ~ list3 ~ Join ~ list4 ~ Join ~ list5 ~ Join ~ list6;
```

We can check the length of the list as follows:

```
Length[list]
```

```
2 000 000
```

Next we want to narrow down our database to only bimolecular reactions, e.g., of the form `A.B>>C` having two reactants and one product. The following code selects only the strings matching that pattern:

```
bimolecularlist =
  Select[list,
   StringMatchQ[#, Except["."] .. ~~ "." ~~ Except["."] .. ~~ ">>" ~~ Except["."] ..] &];
```

Now checking the length of that list:

```
Length[bimolecularlist]
```

```
753 113
```

We're only interested in reactions exclusively involving atoms in the organic subset. The following code selects only reaction strings free of atoms not in the organic subset:

```
reducedsetlist = Select[bimolecularlist,
   StringFreeQ[#, {"Rb", "Cs", "Fr", "Sr", "Ba", "Ra", "Sc", "Ti", "V", "Cr",
      "Mn", "Fe", "Co", "Ni", "Cu", "Zn", "Y", "Zr", "Nb", "Mo", "Tc", "Ru",
      "Rh", "Pd", "Ag", "Cd", "Lu", "Hf", "Ta", "W", "Re", "Os", "Ir", "Pt",
      "Au", "Hg", "Al", "Ga", "In", "Tl", "Si", "Ge", "Sn", "Pb", "As", "Sb",
      "Bi", "Se", "Te", "Po", "At", "He", "Ne", "Ar", "Kr", "Xe", "Rn", "La",
      "Ce", "Pr", "Nd", "Pm", "Sm", "Eu", "Gd", "Tb", "Dy", "Ho", "Er", "Tm",
      "Yb", "Ac", "Th", "Pa", "U", "Np", "Pu", "Am", "Cm", "Bk", "Cf", "Es",
      "Fm", "Md", "No"}] &];
```

Again checking the length of the list:

```
Length[reducedsetlist]
```

```
618 097
```

We are interested in reactions involving complex molecular species, large molecules with complex interactions. So the following code is used to select SMILES reaction strings having a length greater than 400:

```
bigreactantslist = Select[reducedsetlist, StringLength[#] > 400 &];
```

Once again checking the length:

```
Length[bigreactantslist]
```

```
3674
```

We shall be taking a hundred reactions from this list in another notebook, so to avoid taking reactions that are very similar and may have been proximal in the database, we randomize their order:

```
randomizedlist = RandomSample[bigreactantslist, Length[bigreactantslist]];
```

We can see what symbols are used in our list (our "alphabet") in the following way:

```
(*Delete[#,0]& replaces the Head with Sequence*)

alphabet = Union[Delete[Characters[randomizedlist], 0]]
```

```
{@, #, %, (, ), -, +, =, [, ], \, >, ., /, 0, 1, 2, 3, 4,
 5, 6, 7, 8, 9, a, B, c, C, F, g, H, I, K, l, M, n, N, o, O, P, r, s, S}
```

Set the current directory to where you would like to export your list, then export it. Here, the file name is `bigrxns.smiles`, which can be downloaded as discussed in Appendix B. The entries in the file are separated by newline characters. The file can then be opened in MarvinView.

```
SetDirectory["/home/james/Desktop/notebooks/lists"];
Export["bigrxns.smiles", randomizedlist, "List"];
```

5

Markovian Text Generator

In this chapter we will demonstrate how to use a Markovian text generator to generate random molecules, actually pseudo-random SMILES strings that represent candidate molecules for the omega algorithm.

In the 1948 landmark paper *A Mathematical Theory of Communication*, Claude Shannon founded the field of information theory and revolutionized the telecommunications industry, laying the groundwork for the Information Age. In this paper, Shannon proposed using a Markov chain to create a statistical model of the sequences of letters in a piece of text. Markov chains are now widely used in machine recognition, information retrieval, data compressionn, and spam filtering. They also have many scientific computing applications including: the genemark algorithm for gene prediction, the Metropolis algorithm for measuring thermodynamic properties, and Google's pagerank algorithm for Web search, and last but not least, generating pseudo-random text.

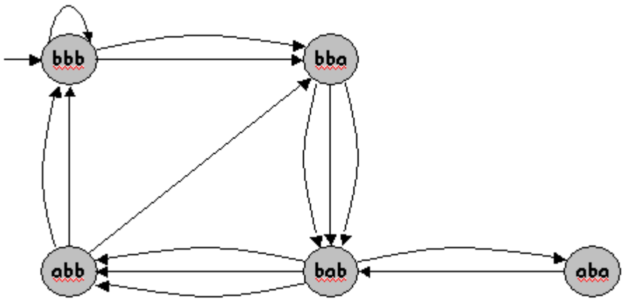
Shannon approximated the statistical structure of a piece of text using a simple mathematical model known as a *Markov model.* A Markov model of order 0 predicts that each letter in the alphabet occurs with a fixed probability. We can fit a Markov model of order 0 to a specific piece of text by counting the number of occurrences of each letter in that text, and using these counts as probabilities. For example, if the input text is "`agggcagcgggcg`", then the Markov model of order 0 predicts that each letter is `a` with probabilty 2/13, `c` with probability 3/13, and `g` with probability 8/13. The following sequence of letters is a typical example generated from this model:

`a g g c g a g g g a g c g g c a g g g g . . .`

An order 0 model assumes that each letter is chosen independently. This does coincide with the statistical properties of English text, for instance, since there is a high correlation among successive letters in an English word or sentence. For example, the letters `h` and `r` and much more likely to follow `t` than either `c` or `x`.

We obtain a more refined model by allowing the probability of choosing each successive letter to depend on the preceding letters or letters. A *Markov model of order n* predicts that each letter occurs with fixed probability, but that probability depends on the previous n consecutive letters (n-gram). For example, if the text has 100 occurrences of `th`, with 60 occurrences of `the`, 25 occurrences of `thi`, 10 occurrences of `tha` and 5 occurrences of `tho`, the Markov model of order 2 predicts that the next letter following the 2-gram `th` is `e` with probability 3/5, `i` with probability 1/4, `a` with probability 1/10 and `o` with probability 1/20.

An approach to generate text according to a Markov model is to create a "Markov chain" and simulate a trajectory through it. A Markov chain is comprised of a set of states, one distinguished state called the start state, and a set of transitions from one state to another. The figure below illustrates a Markov chain with 5 states and 14 transitions.



To simulate a trajectory through the Markov chain, begin at the start state. At each step select one of the leaving arcs uniformly at random, and move to the neighboring state. For example, if the Markov chain is in state `bab`, then it will transition to state `abb` with probability 3/4 and to state `aba` with probability $1/4$. The following are the first seven steps of a possible trajectory beginning from bbb:

bbb→bbb→bba→bab→abb→bbb→bba

A pseudo-random text can be generated according to a Markov model of order n by building a Markov chain from a piece of text, and simulating a trajectory through the Markov chain. A state is included for each of the distinct n-grams that occur in the text. N transitions are included, one from each n-gram in the text to the next overlapping n-gram. Text is treated as a cyclic string to handle boundary cases. For example, if $n = 3$ and the text consists of bbbabbabbbbaba, then the first two transitions to add are from bbb to bba and from bba to bab, and the last (cyclic) one is from abb to bbb. The full Markov chain for $n = 3$ is illustrated in the previous figure.

These ideas are implemented in the python script generate_markov_text.py, which is available for download as discussed in Appendix B. The code for the program is presented in Appendix A.

This script generates pseudo-random text using a Markov chain model. The Markov chain encodes the statistical properties of a source text with a user-specified n-gram length.

The script can be used with two kinds of sources:

1. A text file from which the Markov chain is built.
2. A text file with a pre-generated Markov chain from a previous run.

The generated text can be printed on the console or written to a file, while the Markov chain can be exported to a file and used later (the source file is no longer needed).

Usage examples:

```
$python generate_markov_text -s input_text.txt -l 7 -n 1000
-o output_file.txt -w markov.txt
```

Generates 1000 letters using a 7-gram and writes the Markov chain to markov.txt.

```
$python generate_markov_text -r markov.txt -n 1000 -c
```

Reads a pre-generated Markov chain from a file and prints 1000 letters on the console.

5.1 SMILES String Generation

Generating pseudo-random molecules (that follow a theme) is a straightforward application of the Markovian text generator. The only complication is the generation of invalid SMILES strings, which are mainly due to mismatched brackets and parentheses. We simply filter out those occurrences. The Mathematica notebook `matchcheck.nb` gives the basics of how to generate SMILES strings using the Markovian text generator.

First, set the current directory to where the input file and the program `generate_markov_text.py` is located.

```
SetDirectory["/home/james/Desktop/notebooks/lists"]
```

```
/home/james/Desktop/notebooks/lists
```

```
FileNames[]
```

```
{bigrxns.smiles, candidates.smiles, CFTRA.smiles,
 CFTRB.smiles, CTTadducts.smiles, CTT.smiles, generated.smiles,
 generate_markov_text.py, markov.txt, reactants.smiles, temp.smiles}
```

We have to cull the generated strings that have mismatched brackets and/or parentheses, so we include this simple function for use later (a more effective pair of functions are used in the actual file, download the file to study those):

```
balancedBracketsAndParenthesesQ[str_String] :=
 StringCount[str, "("] === StringCount[str, ")"] &&
  StringCount[str, "("] === StringCount[str, ")"]
```

Next we run the Markov generator to produce a long list of strings:

```
(*run markov generator*)
```

```
Run[
 "python generate_markov_text.py -s CTTadducts.smiles -l 20 -n 1000000 -o
   generated.smiles -w markov.txt"]
```

```
0
```

We import the output file `generated.smiles` as a `List`.

```
listA = Import["generated.smiles", "List"];
```

```
Length[listA]
```

```
2868
```

Some of the generated strings don't include a nucleotide sequence, just an amino acid sequence. Thus, we select on the following basis to only include molecules that *do* include a nucleotide sequence.

```
listA =
  Select[listA,
   StringMatchQ[#, Except["."] .. ~~
      "OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@
        H]1CO" ~~ Except["."] ..] &];
```

We get rid of any duplicate strings:

```
listA = DeleteDuplicates[listA];
```

```
Length[listA]
```

```
545
```

```
listB = Select[listA, balancedBracketsAndParenthesesQ[#] &];
```

```
Length[listB]
```

```
44
```

```
Export["generated.smiles", listB, "List"];
```

The file `generated.smiles` can now be viewed with MarvinView.

6

Cystic Fibrosis

Cystic fibrosis, or CF, is an inherited disease of the secretory glands. People who have CF inherit two faulty genes for the disease – one from each parent. CF mainly affects the lungs, pancreas, liver, intestines, sinuses and sex organs. If you have CF, your mucus becomes thick and sticky. It builds up in your lungs and blocks your airways. The buildup of mucus makes it easy for bacteria to grow. This leads to repeated, serious lung infections. Over time, these infections can severely damage your lungs.

Lung function often starts to decline in early childhood in people who have CF. Over time, damage to the lungs can cause severe breathing problems. Respiratory failure is the most common cause of death in people who have CF. Often, lung transplantation is required when the disease reaches an advanced stage.

In this book, we present a computational technique (the omega algorithm) that can help scientists find cures for hereditary diseases such as CF. To "cure" such a disease means to transform the defective gene to the normal allelic variant. The omega algorithm assesses whether a substance *could* produce the desired transformation of the defective gene to the normal gene. If the omega algorithm determines that a substance could produce the desired change in the gene, the substance is said to pass the *omega test*. Passing the omega test is a necessary but not-necessarily-sufficient condition that needs to be met in order for the substance to have the desired effect. Passing the omega test does not mean the substance is the cure. However, the necessary but not-necessarily-sufficient condition allows scientists to traverse chemical space far more efficiently in the search for a cure. Through the use of the omega algorithm, the cure for CF and other genetic diseases could be found thousands of times more quickly, speeding research towards the goal.

CFTR: The Gene Associated with Cystic Fibrosis
Official Gene Symbol: CFTR
Name of Gene Product: cystic fibrosis transmembrane conductance regulator
Locus: 7q31.2 - The CFTR gene is found in region q31.2 on the long (q) arm of human chromosome 7.
Gene Structure: The normal allelic variant for this gene is about 250,000 base pairs (bp) long and contains 27 exons.
mRNA: The intron-free mRNA transcript for the CFTR gene is 6129 bp long. See the NCBI sequence record `NM_000492` to access the mRNA sequence data.
Coding Sequence (CDS): 4443 bp within the mRNA code for the amino acid sequence of the gene's protein product.
Protein Size: The CFTR protein is 1480 amino acids long and has a molecular weight of 168,173 Da.
Protein Function: The normal CFTR protein product is a chloride channel protein found in membranes of cells that line passageways of the lungs, liver, pancreas, intestines, reproductive tract and skin. CFTR is also involved in the regulation of other transport pathways.
Associated Disorders: Defective versions of this protein, caused by CFTR gene mutations, can lead to the development of cystic fibrosis (CF) and congenital bilateral aplasia of the vas deferens (CBAVD).

6.1 Protein Structure

CFTR is a type of protein classified as an ABC (ATP-binding cassette) transporter or traffic ATPase. These proteins transport molecules such as sugars, peptides, inorganic phosphate, chloride and metal cations across the cellular membrane. CFTR transports chloride ions (Cl^-) across the membranes of cells in the lungs, liver, pancreas, digestive tract, reproductive tract and skin.

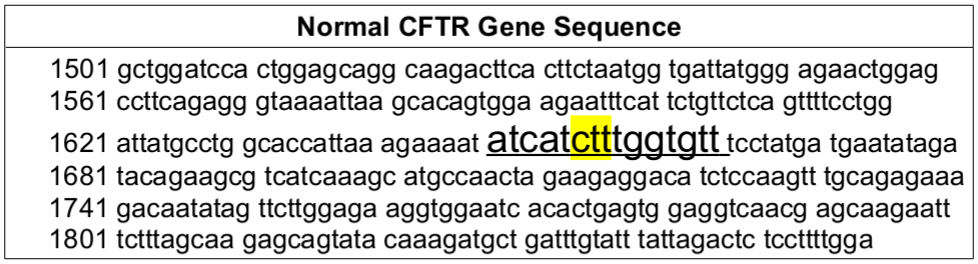
The structure of the complete CFTR protein has not yet been experimentally determined. This is because membrane proteins, such as CFTR, with substantial hydrophobic regions are extremely difficult to crystallize, and X-ray crystallography can only be carried out on protein crystals. By comparing the CFTR protein sequence with that of other known ABC transporters, models depicting the structure of CFTR have been proposed.

CFTR is made up of five domains: two membrane-spanning domains (MSD1 and MSD2) that form the chloride ion channel, two nucleotide-binding domains (NBD1 and NBD2) that bind and hydrolyze ATP (adenosine triphosphate), and a regulatory (R) domain. ΔF508, the most common CF-causing mutation, occurs in the DNA sequence that codes for the first nucleotide-binding domain (NBD1).

While most ABC transporters consist of four domains (two membrane-spanning and two nucleotide-binding domains), CFTR is the only one known to possess a regulatory domain. Modification of the regulatory domain, either through the addition or removal of phosphate groups, has been shown to regulate the movement of chloride ions across the membrane.

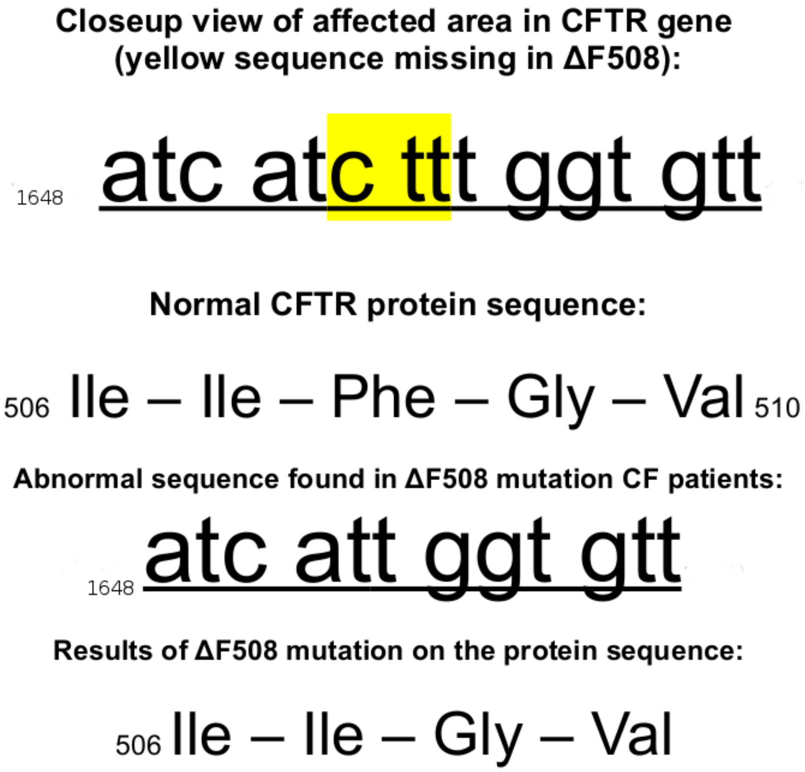
6.2 Common Disease Causing Mutation

About 70% of mutations observed in CF patients result from deletion of three base pairs in CFTR's nucleotide sequence. This deletion causes loss of the amino acid phenylalanine located at position 508 in the protein; therefore this mutation is referred to as Delta F508 or ΔF508.



Normal CFTR Gene Sequence

1501 gctggatcca ctggagcagg caagacttca cttctaatgg tgattatggg agaactggag
1561 ccttcagagg gtaaaattaa gcacagtgga agaatttcat tctgttctca gttttcctgg
1621 attatgcctg gcaccattaa agaaaat atcatctttggtgtt tcctatga tgaatataga
1681 tacagaagcg tcatcaaagc atgccaacta gaagaggaca tctccaagtt tgcagagaaa
1741 gacaatatag ttcttggaga aggtggaatc acactgagtg gaggtcaacg agcaagaatt
1801 tctttagcaa gagcagtata caaagatgct gatttgtatt tattagactc tccttttgga



Closeup view of affected area in CFTR gene
(yellow sequence missing in ΔF508):

1648 atc atc ttt ggt gtt

Normal CFTR protein sequence:

506 Ile – Ile – Phe – Gly – Val 510

Abnormal sequence found in ΔF508 mutation CF patients:

1648 atc att ggt gtt

Results of ΔF508 mutation on the protein sequence:

506 Ile – Ile – Gly – Val

With normal CFTR, once the protein is synthesized, it is transported to the endoplasmic reticulum (ER) and Golgi apparatus for additional processing before being integrated into the cell membrane. When a CFTR protein with the ΔF508 mutation reaches the ER, the quality-control mechanism of this cellular component recognizes that the protein is folded incorrectly and marks the defective protein for degradation. As a result, ΔF508 never reaches the cell membrane.

People who are homozygous for the ΔF508 mutation tend to have the most severe symptoms of cystic fibrosis due to critical loss of chloride ion transport. This upsets the sodium and chloride ion balance needed to maintain the normal, thin mucus layer that is easily removed by cilia lining the lungs and other organs. The sodium and chloride ion imbalance creates a thick, sticky mucus layer that cannot be removed by cilia and traps bacteria, resulting in chronic infections. While the mechanism that leads to lung damage is not fully understood, lung disease is the leading cause of morbidity and mortality among CF patients.

6.3 Factors That Affect the Disease Phenotype

Because CF is an autosomal recessive genetic disorder, an individual must have two copies of a mutated CFTR gene to express the disease phenotype. Someone with one normal, functional copy of the CFTR gene and one mutated copy would just be a carrier of the disorder, and would not display typical CF symptoms. It is important to note that just because two people might have the same two copies of the mutated CFTR gene, each may experience very different symptoms. This is becase the development of a disorder such as CF is greatly influenced by environmental factors and genetic factors other than CFTR.

According to the Cystic Fibrosis Mutation Database maintained by the Hospital for Sick Children in Toronto, Ontario, more than 900 mutations are known in known in the CFTR gene. Different CFTR mutations result in different disease phenotypes. Some may have little or no effect on CFTR function, and some may result in milder forms of disease. For example, one study in the Netherlands indicated that CF patients who had one copy of ΔF508 and one of A455E mutation generally expressed a milder form of pulmonary disease than those who were homozygous ΔF508.

The presence of variant forms of genes other than CFTR can also affect disease phenotype. Meconium ileus (MI), a severe intestinal obstruction, is observed in 15% to 20% of babies born with cystic fibrosis. No CFTR gene mutations have been associated with MI. A 1999 study has shown that cystic fibrosis modifier 1 (CFM1), a modifier gene located on chromosome 19, may determine MI susceptibility.

6.4 Towards a Cure for Cystic Fibrosis

What exactly would it mean to *cure* cystic fibrosis? To *cure* this disease, means to transform the faulty gene responsible for the disease to the normal variant – to change the DNA sequence. How are we to accomplish that? A pharmacological agent must be found that is small enough to enter the cell and nucleus, that is impervious to the action of nucleases, that does not illicit an immune response, that can selectively recognize the ΔF508 DNA sequence, and inserts a CTT oligomer at precisely the right location (between bases 1652 and 1653) in the correct orientation.

In this work we will attempt to find substances that may transform a subsequence of the ΔF508 gene to the normal sequence. This subsequence is comprised of ten bases proximal to the CTT deletion on both the $5'$ and $3'$ ends. This allows for great specificity as $4^{20} > 10^{12}$. We shall refer to the reactant and product subsequences as CFTRA and CFTRB, respectively, and the change we are trying to achieve can be denoted CFTRA→CFTRB. The genetic codes for these subsequences are given by:

$$\textbf{CFTRA:}\ \mathrm{AAAATATCAT}_{\wedge}\mathrm{TGGTGTTTCC}$$

$$\textbf{CFTRB:}\ \mathrm{AAAATATCAT}\underbrace{\mathrm{CTT}}_{\text{insert}}\mathrm{TGGTGTTTCC}$$

As input to our computational procedure, the omega algorithm, we need SMILES string representations of substances that have a fighting chance of producing the transformation (CFTRA.substance)→(CFTRB) when combined with the initial sequence. Both sides of the proposed reaction equation, (CFTRA.substance) and (CFTRB), are evaluated by the omega algorithm to see if they equate. If so, the substance is said to have passed the omega test. For our reasonable input, we shall try CTT-peptide adducts as shown in Fig. 6.1. A list of a variety of CTT-peptide adducts is contained in the file `CTTadducts.smiles` which is used as input to our markovian text generator to generate a vast array of novel molecules that are variations on that general theme. The list can be viewed by opening the file `CTTadducts.smiles` in `MarvinView`. See Appendix B for the address to download the file.

The SMILES string representations for CFTRA and CFTRB were simply generated by drawing the entire molecules into `MarvinSketch`, using its nucleoside templates feature, connecting them with charged phosphate groups, deleting the $2'$ hydroxys, etc., and saving the structures as SMILES strings. The SMILES strings for these two DNA sequences are given in the files `CFTRA.smiles` and `CFTRB.smiles`, respectively, and are available for download as discussed in Appendix B.

The omega algorithm is implemented in the Mathematica notebook file `predict.nb`, which is the subject of the next chapter. With it, it is possible to drastically reduce the size of chemical space that must be experimentally searched in order to find the cure for a heritable disease.

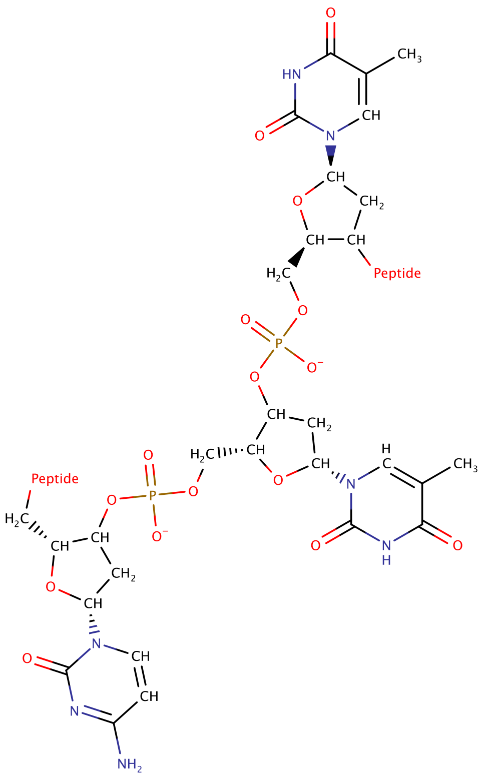


Figure 6.1: A general schematic of the CTT-peptide adducts listed in the input file to the markovian text generator used to generate candidate molecules.

The omega test can be made as stringent as one would like, making it more difficult to pass. This has advantages as low-stringency tests result in more false positives, whereas high-stringency tests give a higher probability that the molecule that passed the test is the actual cure sought after. However, if a high-stringency test is used, far more molecules have to be omega-tested to find an equivalent number of molecules that pass the test.

One might imagine that research towards a cure could be speeded up a thousand-fold through the adoption of this one computational technique. The omega test is in effect a necessary but not-necessarily-sufficient condition that is applied in order to determine if a substance warrants further testing; it does not give definitive answers, so experimentation is still necessary in order to determine if a compound does what it's wanted to do. It is my hope that the readers of this book will improve upon the work I present on this algorithm and further develop it into an even more powerful tool.

7

Omega Algorithm

The *omega algorithm* is a procedure for narrowing down the volume of chemical space that must be experimentally searched in the hunt for a substance that transforms an initial strand of DNA to a desired product strand. Substances that are evaluated to potentially enact the desired transformation are said to have passed the *omega test*. We apply the omega algorithm in this chapter to find molecules that pass the omega test for transformation of the ΔF508 twenty-base subsequence CFTRA to the normal subsequence CFTRB. The two sides of the SMILES reaction string `CFTRA.candidate>>CFTRB` are evaluated to see if they evaluate to the same number. We use the Mathematica function `Predict` to form an abstract model of the chemical reactivity data from part of `bigrxns.smiles`, and then use the model to make predictions regarding the string `CFTRA.candidate`.

The 1st-order omega algorithm is implemented in the notebook `predict.nb`. The first step is to set your current directory to the directory containing the needed files (change the directory in quotes to your own directory).

```
SetDirectory["/home/james/Desktop/notebooks/lists"]
```

```
/home/james/Desktop/notebooks/lists
```

Get a list of the files in the current directory:

```
FileNames[]
```

```
{bigrxns.smiles, candidates.smiles, CFTRA.smiles,
 CFTRB.smiles, CTTadducts.smiles, CTT.smiles, generated.smiles,
 generate_markov_text.py, markov.txt, reactants.smiles, temp.smiles}
```

Next we import the list of reactions from which we will form our abstract chemical reactivity model:

```
rxnlist = Import["bigrxns.smiles", "List"];
```

```
Length[rxnlist]
```

```
3674
```

We split each string into left and right sides, reactants and products, using `>>` as the token. This results in a list of two-member lists. Remember that `bigrxns.smiles` contained only reaction strings that have two reactants and one product.

```
rxnlist = StringSplit[#, ">>"] & /@ rxnlist;
```

For example, the third member of `rxnlist` is now:

```
rxnlist[[3]]
```

```
{[H][C@@]1(O[C@@]2([H])[C@]([H])(O[C@@]3([H])CC[C@@]4(C)[C@@]([H])(CC[C@]5(C)[C@]4([H]
   )C(=O)C=C4[C@]6([H])C[C@](C)(CC[C@]6(C)CC[C@@]54C)C(O)=O)C3(C)C)O[C@]([H])(C(O)=O)
   [C@@]([H])(O)[C@]2([H])O)OC[C@](O)(CO)[C@@]1([H])O.C=[N+]=[N-],
 [H][C@@]1(O[C@@]2([H])[C@]([H])(O[C@@]3([H])C4C[C@@]5(C)[C@@]([H])(CC[C@]6(C)[C@]5([H
   ])C(=O)C=C5[C@]([H])(C[C@](C)(CC4)C(=O)OC)[C@@]([H])(C)CC[C@@]65C)C3(C)C)O[C@]([H]
   )(C(=O)OC)[C@@]([H])(O)[C@]2([H])O)OC[C@](O)(CO)[C@@]1([H])OC}
```

One can hardcode the SMILES strings for CFTRA and CFTRB as follows, but one could `Import` the files `CFTRA.smiles` and `CFTRB.smiles`.

```
CFTRA =
  "CC1=CN([C@H]2CC(OP([O-])(=O)OC[C@H]3O[C@H](CC3OP([O-])(=O)OC[C@H]3O[C@H](CC3O)N3C=CC
    (N)=NC3=O)N3C=CC(N)=NC3=O)[C@@H](COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3CO)n3cnc4c(N)ncnc34)n3cnc4c(N)ncnc34)n3cnc4c(N)ncnc34)n3cnc4c(
    N)ncnc34)N3C=C(C)C(=O)NC3=O)n3cnc4c(N)ncnc34)N3C=C(C)C(=O)NC3=O)N3C=CC(N)=NC3=O)
    n3cnc4c(N)ncnc34)N3C=C(C)C(=O)NC3=O)N3C=C(C)C(=O)NC3=O)n3cnc4c3N=C(N)NC4=O)
    n3cnc4c3N=C(N)NC4=O)N3C=C(C)C(=O)NC3=O)n3cnc4c3N=C(N)NC4=O)N3C=C(C)C(=O)NC3=O)N3C
    =C(C)C(=O)NC3=O)O2)C(=O)NC1=O";
```

```
CFTRB =
  "CC1=CN([C@H]2CC(OP([O-])(=O)OC[C@H]3O[C@H](CC3OP([O-])(=O)OC[C@H]3O[C@H](CC3O)N3C=CC
    (N)=NC3=O)N3C=CC(N)=NC3=O)[C@@H](COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)
    OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3COP([O-])(=O)OC3C[C@@H](O[C@@H
    ]3COP([O-])(=O)OC3C[C@@H](O[C@@H]3CO)n3cnc4c(N)ncnc34)n3cnc4c(N)ncnc34)n3cnc4c(N)
    ncnc34)n3cnc4c(N)ncnc34)N3C=C(C)C(=O)NC3=O)n3cnc4c(N)ncnc34)N3C=C(C)C(=O)NC3=O)
    N3C=CC(N)=NC3=O)n3cnc4c(N)ncnc34)N3C=C(C)C(=O)NC3=O)N3C=CC(N)=NC3=O)N3C=C(C)C(=O)
    NC3=O)N3C=C(C)C(=O)NC3=O)N3C=C(C)C(=O)NC3=O)n3cnc4c3N=C(N)NC4=O)n3cnc4c3N=C(N)NC4
    =O)N3C=C(C)C(=O)NC3=O)n3cnc4c3N=C(N)NC4=O)N3C=C(C)C(=O)NC3=O)N3C=C(C)C(=O)NC3=O)
    O2)C(=O)NC1=O";
```

Later when we use the Markovian text generator, we will need the following function to cull invalid SMILES strings the generator produces (a more advanced pair of functions is used in the actual file).

```
balancedBracketsAndParenthesesQ[str_String] :=
 StringCount[str, "["] === StringCount[str, "]"] &&
  StringCount[str, "("] === StringCount[str, ")"]
```

Now we form the abstract chemical reactivity model. We form a training set to feed to the `Predict` function. For each reaction, the left and right sides are trained to evaluate to the same unique number. The training set cannot be too large, or too small. Here, we train on a data set of 100 reactions, and also set `CFTRA` and `CFTRB` to arbitrary, but different, numbers. This is the initialization phase. Later, we reset the number `CFTRB` is set to in order to overcome the “can’t get there from here” phenomenon that can occur in this computation.

```
(* initialization - train, predict, find rare occuring number,
then retrain with CFTRB set to that number *)

trainingset = Join[Table[rxnlist[[i, 1]] → i, {i, n = 100}],
   Table[rxnlist[[i, 2]] → i, {i, n}], {CFTRA → 1}, {CFTRB → 100}];

Length[trainingset]

202
```

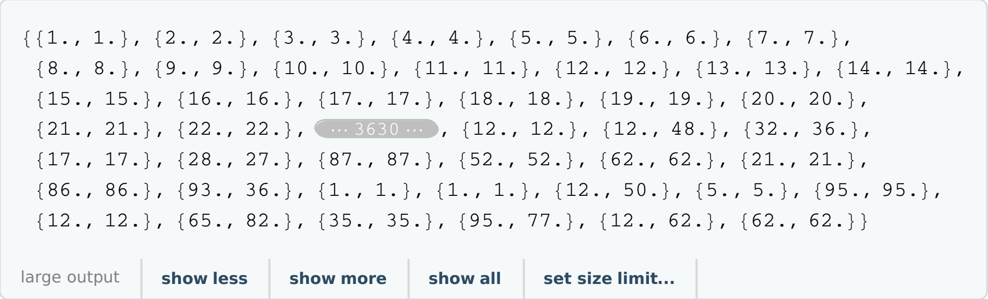
The next step is to compute our initial abstract model. We use the machine learning function Predict to do this, and choose "NearestNeighbors" as our Method.

```
p = Predict[trainingset, Method -> "NearestNeighbors"]
```

```
PredictorFunction[ Method: NearestNeighbors
                   Feature type: Text ]
```

The PredictorFunction thereby produced can then be mapped across the left and right sides of each reaction in rxnlist to see how effective the learning was.

```
Map[p, rxnlist, {2}]
```



```
{{1., 1.}, {2., 2.}, {3., 3.}, {4., 4.}, {5., 5.}, {6., 6.}, {7., 7.},
 {8., 8.}, {9., 9.}, {10., 10.}, {11., 11.}, {12., 12.}, {13., 13.}, {14., 14.},
 {15., 15.}, {16., 16.}, {17., 17.}, {18., 18.}, {19., 19.}, {20., 20.},
 {21., 21.}, {22., 22.}, ··· 3630 ···, {12., 12.}, {12., 48.}, {32., 36.},
 {17., 17.}, {28., 27.}, {87., 87.}, {52., 52.}, {62., 62.}, {21., 21.},
 {86., 86.}, {93., 36.}, {1., 1.}, {1., 1.}, {12., 50.}, {5., 5.}, {95., 95.},
 {12., 12.}, {65., 82.}, {35., 35.}, {95., 77.}, {12., 62.}, {62., 62.}}
```

Remember that we used only the first 100 reactions for training data. One can see from above that the training data was learned perfectly, whereas roughly 60% of the untrained reaction pairs have equivalent values for the left and right SMILES strings. This indicates that the system learned something about how the strings transform, and this is therefore a type of string transformation learning.

We can check the values that CFTRA and CFTRB evaluate to.

```
p[CFTRA]
```

```
1.
```

```
p[CFTRB]
```

```
100.
```

We need some novel molecules to omega-test. The first step is to generate a Markov chain and save it to a file, `markov.txt`. We can execute the Markov generator `generate_markov_text.py` command-line statement from within Mathematica itself. Mathematica returns 0 when the program finishes.

```
(* run markov generator *)

Run["python generate_markov_text.py -s CTTadducts.smiles -l 20 -w markov.txt"]
```

```
0
```

Next we generate some random molecules, SMILES strings, some of which will be invalid.

```
Run["python generate_markov_text.py -r markov.txt -n 10000000 -o generated.smiles"]
```

```
0
```

Now `Import` that list of strings and check its length.

```
listA = Import["generated.smiles", "List"];
```

```
Length[listA]
```

```
28 862
```

Some of the generated molecules will be peptides, with no nucleotides, which would not be reasonable candidates to effect our desired transformation. Using the find function in a text editor revealed the presence of a common sequence found in the SMILES strings of compounds containing the CTT moeity. We select for those strings.

```
listA =
  Select[listA,
   StringMatchQ[#, Except["."] .. ~~
      "OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@
        H]1CO" ~~ Except["."] ..] &];
```

We can easily get rid of any duplicate strings as follows:

```
listA = DeleteDuplicates[listA];
```

```
Length[listA]
```

```
4821
```

Next we need to get rid of strings having mismatched brackets and/or parentheses. A simple way is as follows:

```
listB = Select[listA, balancedBracketsAndParenthesesQ[#] &];
```

```
Length[listB]
```

```
325
```

Now we are going to combine the SMILES string for `CFTRA` with that of each candidate substance, and also react the substances by inserting a period between the SMILES strings.

```
listC = StringJoin[CFTRA, ".", #] & /@ listB;
```

Now to see if the combined strings evaluate to the value of `CFTRB` (100).

```
listD = p[#] & /@ listC
```

```
{1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 5., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 5., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 62., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 62., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 62., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 12., 1., 1., 1., 1., 62., 1., 1., 1., 62., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 62., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}
```

```
Length[listD]
```

```
325
```

One can see that many strings evaluated to 1 (=CFTRA indicating no reaction). None of them evaluated to CFTRB=100. In fact, there is a “can’t get there from here” phenomenon that occurs in this computation which can only be overcome be reseting the value of CFTRB to an obtainable number and retraining. Here we shall opt for a low-stringency test and set the value of CFTRB to 62. But a higher-stringency test could be used by resetting the value of CFTRB to 5 or 12. The entire reactivity model is perturbed.

```
(* retrain *)

trainingset = Join[Table[rxnlist[[i, 1]] → i, {i, n = 100}],
   Table[rxnlist[[i, 2]] → i, {i, n}], {CFTRA → 1}, {CFTRB → 62}];

Length[trainingset]
```

```
202
```

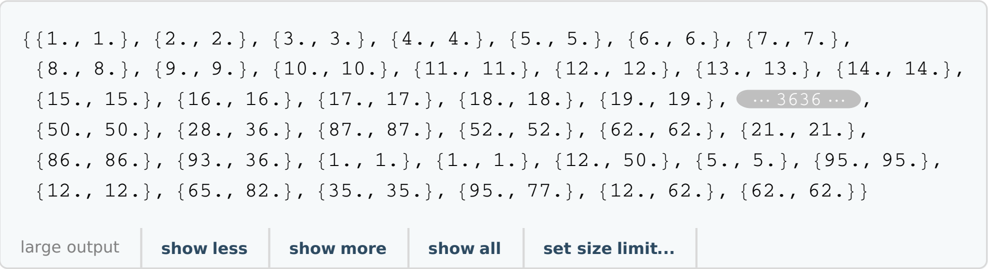
```
p = Predict[trainingset, Method -> "NearestNeighbors"]
```

```
PredictorFunction[ Method: NearestNeighbors
                   Feature type: Text ]
```

```
(* end of initialization *)
```

Again checking how well the map was learned:

```
Map[p, rxnlist, {2}]
```



```
{{1., 1.}, {2., 2.}, {3., 3.}, {4., 4.}, {5., 5.}, {6., 6.}, {7., 7.},
 {8., 8.}, {9., 9.}, {10., 10.}, {11., 11.}, {12., 12.}, {13., 13.}, {14., 14.},
 {15., 15.}, {16., 16.}, {17., 17.}, {18., 18.}, {19., 19.}, ··· 3636 ··· ,
 {50., 50.}, {28., 36.}, {87., 87.}, {52., 52.}, {62., 62.}, {21., 21.},
 {86., 86.}, {93., 36.}, {1., 1.}, {1., 1.}, {12., 50.}, {5., 5.}, {95., 95.},
 {12., 12.}, {65., 82.}, {35., 35.}, {95., 77.}, {12., 62.}, {62., 62.}}
```

```
p[CFTRA]
```

```
1.

p[CFTRB]

62.

PCFTRA = p[CFTRA];
PCFTRB = p[CFTRB];

(* run markov generator *)

Run["python generate_markov_text.py -s CTTadducts.smiles -l 20 -w markov.txt"]

0


Run["python generate_markov_text.py -r markov.txt -n 10000000 -o generated.smiles"]

0

listA = Import["generated.smiles", "List"];

Length[listA]

28 860

listA =
  Select[listA,
   StringMatchQ[#, Except["."] .. ~~
      "OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@
        H]1CO" ~~ Except["."] ..] &];

listA = DeleteDuplicates[listA];

Length[listA]

4857

listB = Select[listA, balancedBracketsAndParenthesesQ[#] &];

Length[listB]

349
```

```
listC = StringJoin[CFTRA, ".", #] & /@ listB;
```

We now apply our new `PredictorFunction` to the reactant strings:

```
listD = p[#] & /@ listC
```

```
{1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 62., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 5., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 62., 1., 62., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 12., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}
```

We can see that our necessary but not-necessarily sufficient condition, the omega test, was passed three times in this example.

```
Length[listD]
```

```
349
```

The positions in the list of the molecules that passed the omega test can be found as such:

```
pos = Position[listD, PCFTRB] // Flatten
```

```
{59, 191, 193}
```

```
candidates = {};
```

And we select the SMILES strings for the omega-test passers this way:

```
candidates = Select[Append[candidates, listB[[pos]]],
    UnsameQ[#, {}] &] // Flatten;
```

```
Length[candidates]
```

```
3
```

We can export the list to a file, then view those molecules with MarvinView.

```
Export["candidates.smiles", candidates, "List"];
```

The process of candidate molecule selection can be put in a loop:

```
(**********************   LOOP   ************************)


candidates = {};
totalevaluated = 0;
cycles = 0;
While[Length[candidates] < 10,
 cycles = cycles + 1;
 Run["python generate_markov_text.py -r markov.txt -n 10000000 -o generated.smiles"];
 listA = Import["generated.smiles", "List"];
 listA =
  Select[listA,
   StringMatchQ[#, Except["."] .. ~~
      "OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@H]1COP([O-])(=O)OC1C[C@@H](O[C@@
        H]1CO" ~~ Except["."] ..] &];
 listA = DeleteDuplicates[listA];
 listB = Select[listA, balancedBracketsAndParenthesesQ[#] &];
 listC = StringJoin[CFTRA, ".", #] & /@ listB;
 listD = p[#] & /@ listC;
 pos = Position[listD, PCFTRB] // Flatten;
 candidates = Select[Append[candidates, listB[[pos]]], UnsameQ[#, {}] &] // Flatten;
 totalevaluated = totalevaluated + Length[listB];
 If[Mod[cycles, 1] == 0 || Length[candidates] > 0 ,
  Print["cycles = ", cycles];
  Print["total molecules evaluated = ", totalevaluated];
  Print["candidate molecules found = ", Length[candidates]];];
]
```

```
Length[candidates]
```

```
13
```

```
Export["candidates.smiles", candidates, "List"];
```

We show an interesting molecule that passed the omega test at low-stringency for correction of the ΔF508 gene to the normal variant in figure 7.1. Note the presence of a CTT oligomeric sequence and two peptide scissor-hands.

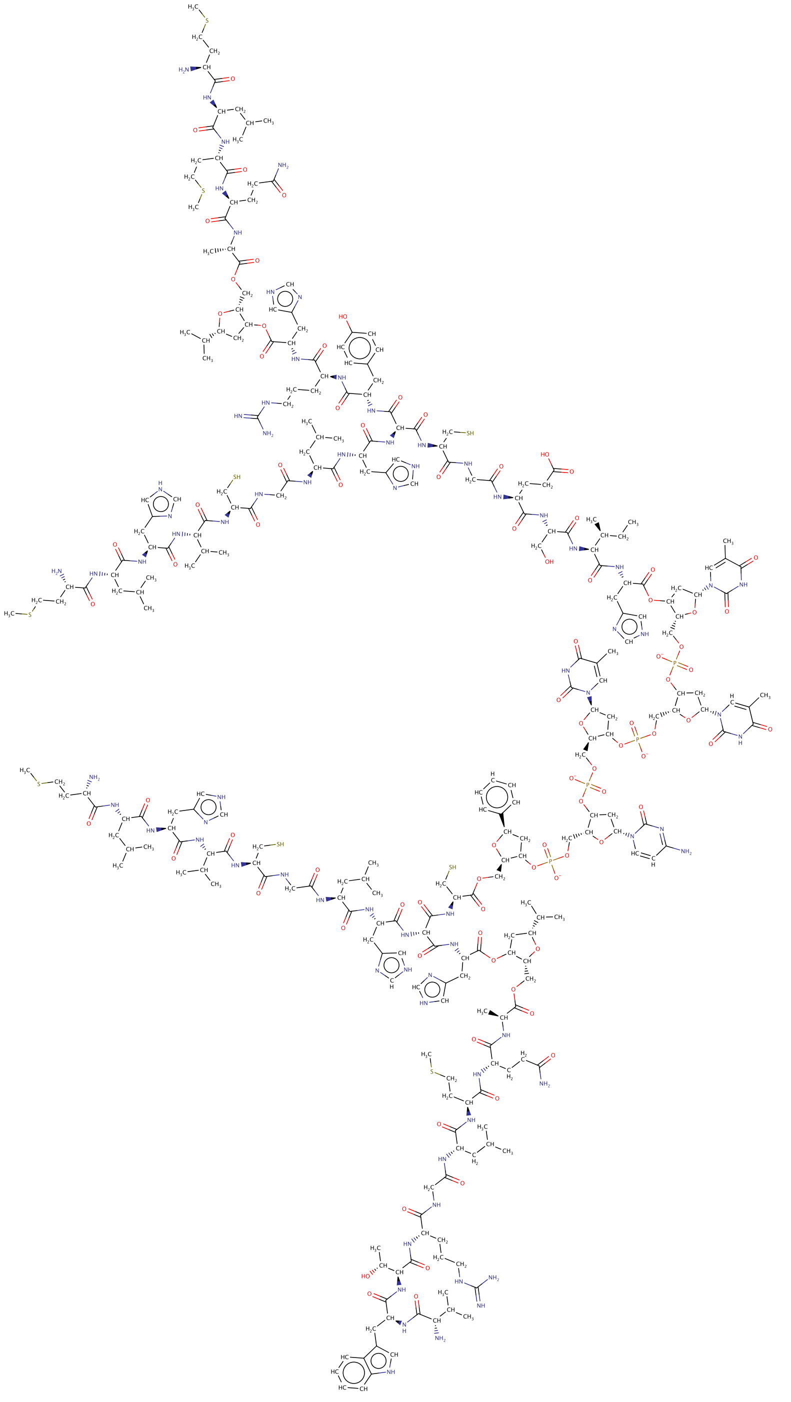


Figure 7.1: An interesting molecule that passed the omega test at low stringency for correction of the ΔF508 gene. Notice the presence of a CTT sequence and two peptide scissor-hands.

The molecule shown in figure 7.1, of course, is probably not the cure for cystic fibrosis, but finding it did demonstrate the omega algorithm in short order. What we've done so far is perform a 1$^{\mathrm{st}}$-order omega test.

How might the omega algorithm be made more definitive in its answers? It would be desirable to experimentally search as little of chemical space as possible in the hunt for the cure. More definitive answers are the subject of *higher-order omega tests*. They are the subject of the next chapter.

8

Extended Omega Algorithm

The omega algorithm can be made more definitive through the consideration of n$^{\text{th}}$-order omega tests. These are similar to first-order omega tests, but involve rather the input of previously outputted candidate molecule SMILES strings as the source for the Markov chain formation, output of new candidates based on that model, in a repetitive loop. We refer to the algorithm involving higher-order omega tests as the extended omega algorithm. An n$^{\text{th}}$-order omega test involves n loops. This algorithm causes the accumulation of structural properties favorable to the transformation.

The extended omega algorithm is implemented in the file `predict2.nb` and the SMILES strings produced after 9 more iterations (10$^{\text{th}}$-order omega test) are located in the file `candidates2.SMILES`.

Figure 8.1 shows a molecule, dubbed F508-001, that passed a 10$^{\text{th}}$-order omega test at high stringency for correction of the (+)-strand of the ΔF508 gene. Notice the presence of a $3'$-CTT-$5'$ sequence and the many nitrogenous bases, nucleotides and riboses intermingled with the peptide sequences. Note that our initial strand CFTRA lacks the subsequences AAG and GAA. This molecule warrants further experimental investigation, which shall be left to the scientific community.

A precise mechanism for how F508-001 might work is unknown, but some initial guesswork as to the spatial layout of the reaction is shown in figure 8.2. The peptide sequences are assumed to play both a catalytic role and a role in DNA sequence recognition.

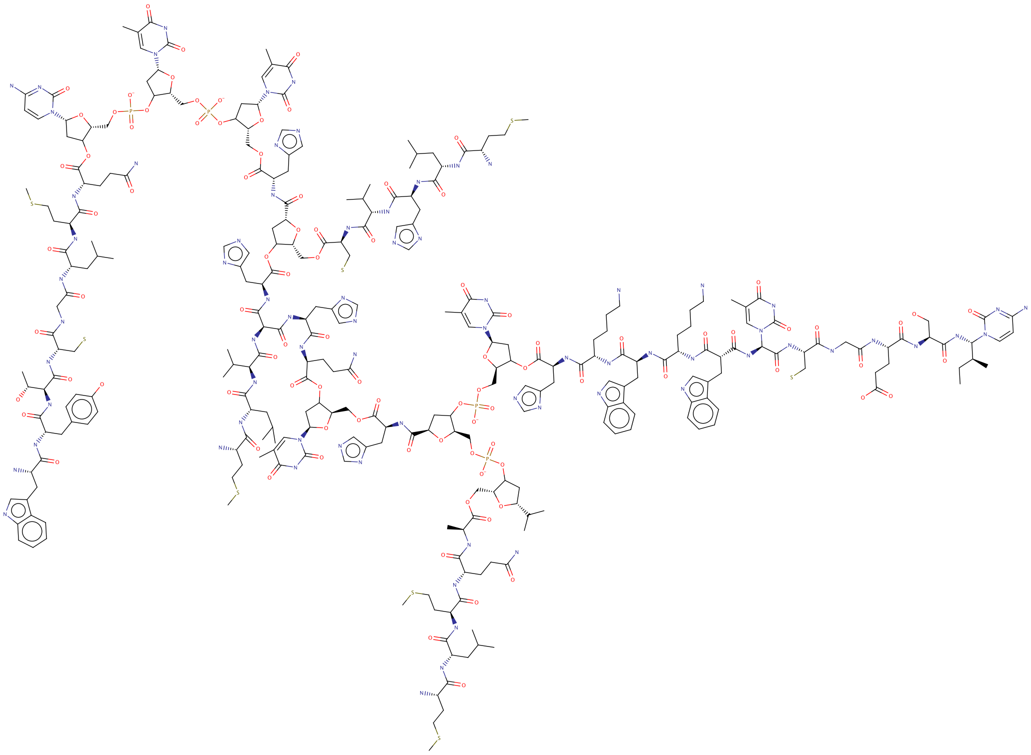


Figure 8.1: The compound F508-001 passed a 10th-order omega test at high stringency for correction of the (+)-strand of the ΔF508 gene. Notice the presence of a $3'$-CTT-$5'$ sequence and the many nitrogenous bases, nucleotides and riboses intermingled with the peptide sequences. Note that our initial strand CFTRA lacks the subsequences AAG and GAA.

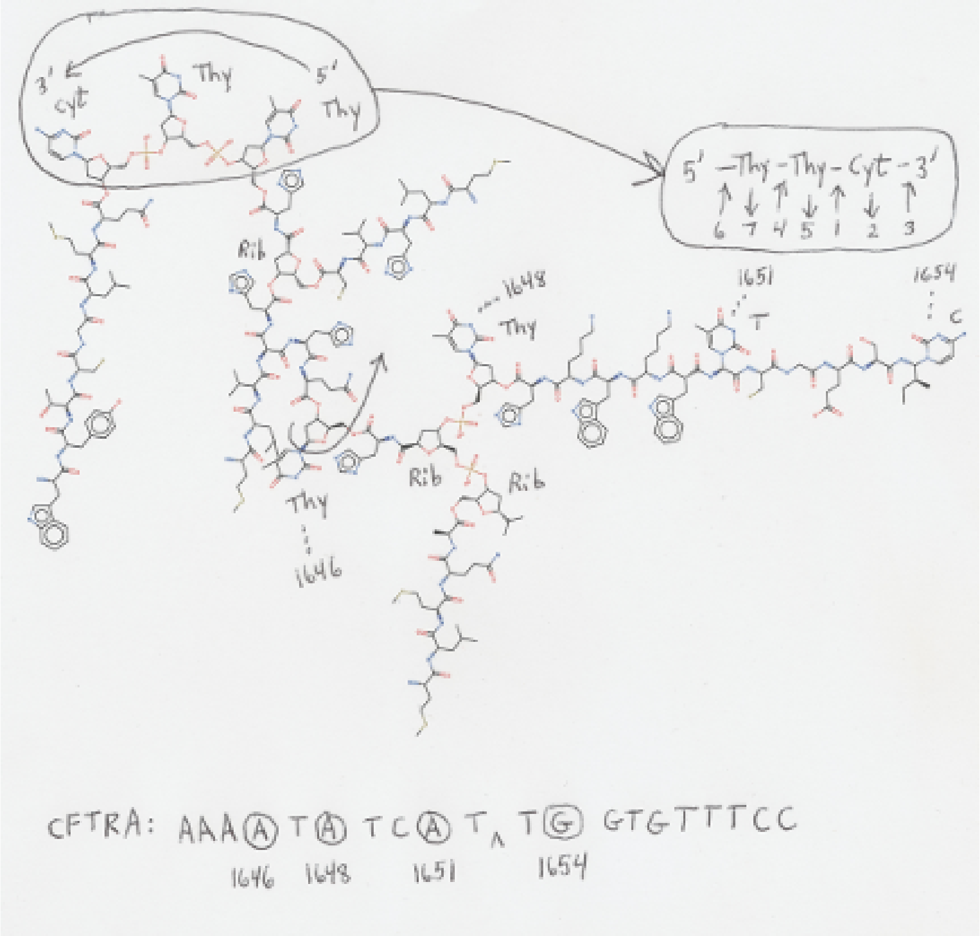


Figure 8.2: A precise mechanism for how F508-001 might work is unknown, but some initial guesswork as to the spatial layout of the reaction is shown here. The peptide sequences are assumed to play both a catalytic role and a role in DNA sequence recognition.

A

generate_markov_text.py

```
# Copyright (c) 2013 Gratian Lup. All rights reserved.
# Redistribution and use in source and binary forms, with or
    without
# modification, are permitted provided that the following
    conditions are
# met:
#
# * Redistributions of source code must retain the above
    copyright
# notice, this list of conditions and the following disclaimer.
#
# * Redistributions in binary form must reproduce the above
# copyright notice, this list of conditions and the following
# disclaimer in the documentation and/or other materials
    provided
# with the distribution.
#
# * The name "MarkovTextGenerator" must not be used to endorse
    or promote
# products derived from this software without prior written
    permission.
#
# * Products derived from this software may not be called "
    MarkovTextGenerator" nor
# may "MarkovTextGenerator" appear in their names without prior
     written
# permission of the author.
#
```

```
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
# A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
# OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
# SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
# LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
# DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
# THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
# OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# This script generates pseudo-random text using a Markov chain model.
# The Markov chain encodes the statistical properties of a source text
# with a user-specified n-gram length.
# More details on the algorithm can be found on the following page:
# http://www.cs.princeton.edu/courses/archive/spr05/cos126/assignments/markov.html
#
# The script can be used with two kind of sources:
# 1. A text file from which the Markov chain is built.
# 2. A text file with a pre-generated Markov chain from a previous run.
#
# The generated text can be output on the console or written to a file,
# while the Markov chain can be exported to a file and used later.
#
# Usage examples:
```

```
#     generate_markov_text -s input_text.txt -l 7 -n 1000 -o
    output_file.txt -w markov.txt
#         generates 1000 letters using a 7 letter n-gram
#         and writes the Markov chain to "markov.txt"
#     generate_markov_text -r markov.txt -n 1000 -c
#         reads the Markov chain from a file and prints 1000
    letters on the console
#
# For usage information execute the script with the --help flag
    .
from random import *
from optparse import *
import string

class MarkovChainNode:
    def __init__(self, ngram):
        self.ngram = ngram
        self.next_states = []

    def add_next_state(self, node, probability):
        self.next_states.append((node, probability))
        self.next_states.sort(key = lambda pair : pair[1],
            reverse = True)

    def get_next_state(self):
        # Randomly select a next node from the chain, giving
            higher
        # priority to the letters that follow this n-gram more
            often.
        probability = random()

        for state in self.next_states:
            if probability < state[1]:
                return state[0]
            else:
                probability -= state[1]

        # The input text is treated like a circular buffer,
        # so all chain nodes should have a next state.
        raise Exception("No next state found for '{0}", self.
            ngram)


def compute_ngram_counts(text, k):
    ngrams = {}
    text_length = len(text)

    if text_length < k:
        # Text is too short to extract anything useful.
        raise Exception("Text is too short. Provide more than
            {0} characters".format(k))

    for i in range(0, text_length):
        # Extract the n-gram from positions [i, i + k).
        # If it extends beyond the text, letters from the
            beginning are taken.
        if i + k < text_length:
            ngram = text[i : i + k]
        else:
            ngram = text[i : text_length] + \
                    text[0 : (k - (text_length - i))]

        # Look at the letter following the n-gram and increase
            the count
        # associated with it. If it is the first time it is
            seen, the count is 1.
        next_letter = text[i + k] if i + k < text_length else \
                      text[k - (text_length - i)]

        # The letters following a n-gram are stored as a
            dictionary of
        # (letter : occurrence_count) pairs.
        if not ngram in ngrams:
            ngrams[ngram] = {}

        next_ngram_letters = ngrams[ngram]

        if next_letter in next_ngram_letters:
            next_ngram_letters[next_letter] += 1
        else:
            next_ngram_letters[next_letter] = 1

    return ngrams


def print_ngram_counts(ngrams):
    for ngram, next_letters in ngrams.items():
        print("n-gram '{0}':".format(ngram))

        for letter, count in next_letters.items():
            print("    {0}: {1}".format(letter, count))


def read_source_file(file_path):
    with open(file_path, "r") as file:
        return file.read()


def string_to_decimal(text):
    return " ".join([str(ord(letter)) for letter in text])


def string_from_decimal(decimal):
    return "".join([chr(int(number)) for number in decimal.
        split()])


def write_ngram_counts_to_file(ngrams, ngram_length, file_path)
    :
    with open(file_path, "w") as file:
        # On the first line write the length and number of n-
            grams,
        # then each n-gram on a separate line.
        file.write("{0} {1}\n".format(ngram_length, len(ngrams)
            ))

        for ngram, next_letters in ngrams.items():
            # Write the n-gram, the letter number and the
                letter-count pairs.
            file.write("{0} {1} ".format(string_to_decimal(
                ngram), len(next_letters)))

            for letter, count in next_letters.items():
                file.write("{0} {1} ".format(string_to_decimal(
                    letter), count))

            file.write("\n")


def read_ngram_counts_from_file(file_path):
    ngrams = {}
    ngram_length = 0

    with open(file_path, "r") as file:
        # Read the length and the number of n-grams.
        ngram_length, ngram_count = [int(number) for number in
            file.readline().split()]

        for i in range(0, ngram_count):
            # The line begins with the n-gram having each
                letter in decimal,
            # followed by the number of letter-count pairs.
            ngram_info = file.readline().split()
            ngram_value = string_from_decimal(" ".join(
                ngram_info[0:ngram_length]))
            next_letter_count = int(ngram_info[ngram_length])

            # Read the letter-count pairs.
            next_letters = {}

            for j in range(0, next_letter_count):
                letter = string_from_decimal(ngram_info[1 +
                    ngram_length + (j * 2)])
                count = int(ngram_info[1 + ngram_length + (j *
                    2 + 1)])
                next_letters[letter] = count

            ngrams[ngram_value] = next_letters

    return (ngrams, ngram_length)


def build_markov_chain(ngrams):
    # First build the Markov nodes for all n-grams,
    # then connect the nodes using the next-letter information.
    chain_nodes = {}

    for ngram in ngrams:
        chain_nodes[ngram] = MarkovChainNode(ngram)

    for ngram, next_letters in ngrams.items():
        # For each letter compute the probability that it
            follows after the n-gram.
        weight = float(sum((count for letter, count in
            next_letters.items())))
        node = chain_nodes[ngram]

        for letter, count in next_letters.items():
            # The next n-gram consists of the first K-1 letters
            # from the current node and the last letter from
                the next node.
            next_state_ngram = ngram[1:] + letter
            next_state_node = chain_nodes[next_state_ngram]
            node.add_next_state(next_state_node, count / weight
                )

    return chain_nodes


def generate_text(chain, length):
    # Randomly select one of the chain nodes as the start node.
    # For the start node the entire n-gram is used, while for
        the next
    # states only the last character, until the required length
        is reached.
    ngrams = [key for key in chain.keys()]
    node = chain[choice(ngrams)]

    text = []
    text_length = len(node.ngram)

    for letter in node.ngram:
        text.append(letter)

    while text_length < length:
        node = node.get_next_state()
        text.append(node.ngram[-1])
        text_length += 1

    return "".join(text)


def write_text_to_file(text, file_path, words_per_line = 12):
    with open(file_path, "w") as file:
        words = 0

        for letter in text:
            if letter in string.whitespace:
                words += 1
                file.write("\n" if (words % words_per_line ==
                    0) else letter)
            else:
                file.write(letter)
```

```
def main():
    parser = OptionParser()
    parser.add_option("-s", "--source", dest = "source",
                      help = "The file containing the text to
                          be analyzed.")
    parser.add_option("-l", "--length", dest = "length",
                      help = "The length of the used n-gram (in
                           letters).")
    parser.add_option("-n", "--number", dest = "number",
                      help = "The number of letters the output
                          text should contain.")
    parser.add_option("-c", "--console", dest = "console",
        action = "store_true",
                      help = "Print the output text on the
                          console.")
    parser.add_option("-o", "--output", dest = "output",
                      help = "The file where the output text
                          should be written.")
    parser.add_option("-r", "--read_markov", dest = "
        read_markov",
                      help = "The file from where to read the
                          Markov chains.")
    parser.add_option("-w", "--write_markov", dest = "
        write_markov",
                      help = "The file where to write the
                          Markov chains.")
    options, args = parser.parse_args()

    # There are two supported work modes:
    # 1. Input text read from a file, followed by building the
        Markov chain.
    # 2. Pre-generated Markov chain read from a file.
    ngrams = None
    ngram_length = None
    output_text = None

    if options.number is None:
        print("Length of output text not specified!")
        return -1
    elif options.console is None and options.output is None:
        print("No type of output specified!")
        return -1

    if options.source is not None:
        if options.read_markov is not None:
            print("Source file and Markov chain file cannot be
                used at the same time!")
            return -1
        elif options.length is None:
            print("Length of used n-gram not specified!")
            return -1

        print("Generating {0} letters from source file {1}". \
              format(options.number, options.source))
        ngram_length = int(options.length)
        text = read_source_file(options.source)
        ngrams = compute_ngram_counts(text, ngram_length)
    else:
        if options.read_markov is None:
            print("A data source must be specified!")
            return -1

        print("Generating {0} letters from Markov chain file
            {1}". \
              format(options.number, options.read_markov))
        ngrams, ngram_length = read_ngram_counts_from_file(
            options.read_markov)

    # Now build the Markov chain and create the text.
    chain = build_markov_chain(ngrams)
    output_text = generate_text(chain, int(options.number))

    if options.console is not None:
        print("Output text: {0}".format(output_text))

    if options.output is not None:
        write_text_to_file(output_text, options.output)

    if options.write_markov is not None:
        write_ngram_counts_to_file(ngrams, ngram_length,
            options.write_markov)

    return 0


if __name__ == '__main__':
    main()
```