Fast and Interpretable Classification of Sequential Data in Biology

---

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by

Ritambhara Singh

May 2018

APPROVAL SHEET

This Dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Author Signature: _____

This Dissertation has been read and approved by the examining committee:

Advisor: Dr. Yanjun Qi

Committee Member: Dr. Mary Lou Soffa

Committee Member: Dr. Gabriel Robins

Committee Member: Dr. Mazhar Adli

Committee Member: Dr. Christina Leslie

Committee Member: _____

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

May 2018

Abstract

Biological sciences are rapidly becoming data intensive. Between 100 million to 2 billion human genomes are estimated to be sequenced by the year 2025, far exceeding the growth of big data domains like Astronomy, YouTube, and Twitter. Majority of these biological datasets are sequential in nature, representing the human genome as well as measurements of activity taking place around it. Analyzing this enormous repository of sequential data is both urgent and essential to understand genetic diseases and drug development. Data-driven approaches like machine learning have shown significant progress in analyzing the existing data. However, the state-of-the-art machine learning techniques face two hard challenges in this domain: (1) Interpretability of the predictions for better insights, and (2) Slow computation due to expanding search space of sequential patterns. In this dissertation, we aim to solve these two challenges by improving two popular machine learning models: Deep Neural Networks (DNNs) and String Kernel with Support Vector Machines (SK-SVM).

Challenge(1): DNNs can handle large sequential datasets accurately and in an efficient manner. However, DNNs have widely been viewed as 'black boxes' due to the complex, multi-layer structure, making them hard to understand. We design a novel unified DNN architecture to model and to interpret features in an end-to-end manner. The proposed design is not only accurate, but it also provides better interpretation than state-of-the-art feature visualization methods such as saliency maps.

Challenge (2): SK-SVM methods achieve high accuracy and have theoretical guarantees with limited labeled training samples. However, current implementations run extremely slow when we increase the dictionary size or allow more mismatches. We present a novel algorithmic implementation for calculating Gapped k-mer string Kernel using Counting (GaKCo). This method is fast, scalable and naturally parallelizable. Empirically, GaKCo is up to 100 times faster than the state-of-the-art SK-SVM method across multiple biological sequential datasets.

Acknowledgements

I would like to start by expressing my utmost gratitude to Dr. Yanjun Qi. She has been an incredible advisor and a mentor for the past five years. She understood my keen interest in doing interdisciplinary research and encouraged my ideas. Her guidance during critical stages of my Ph.D. has not only shaped my research but also helped me figure out a career path.

I would also like to thank Dr. Mazhar Adli for accepting me as a student in his lab and being incredibly patient as I started working with biological datasets. His insights regarding my projects were beneficial towards their development and execution. I am fortunate to have a great dissertation committee including Dr. Mary Lou Soffa, Dr. Gabriel Robins, and Dr. Christina Leslie and I am grateful for their valuable inputs.

I have had some wonderful mentors during different stages of my Ph.D. Dr. Ryan Layer, who was a graduate student when I started, shared my passion for biology. He connected me to the labs in the Medical School that began this journey. Royden Clark patiently showed me the ropes of data analysis in bioinformatics and was always there to help me when I got stuck on a problem. I also had the excellent opportunity to collaborate with Dr. Jennifer Listgarten. She reminded me of the importance of curiosity and unapologetically asking questions in research.

No woman is an island, and hence I owe this milestone to an outstanding support system of family and friends. My parents and my sister have been my pillars of support and encouragement. My uncle, Aditya, aunt, Kiran, and their beautiful family has been my home away from home. They went above and beyond to help me settle here and have been my safety net. The first group of friends at a new place are paramount to help one adjust well, and I have been lucky to have quite a few. A special mention to Andrew, Debalina, Jacob, Jeremy, Juhi, and Vidhya for being few of those amazing friends. Avinash and Divya are my rockstars. They

cheered me on along every step of the way that restored my confidence in my abilities. Avinash introduced me to backpacking, while Divya got me into rock climbing, and both these hobbies have helped me develop as a person. Jack has been a collaborator and a close friend, with whom I have bounced project ideas and occasionally pondered over the meaning of life. Ivan has been an ally since day one and our monthly lunches and interesting conversations would be my highlights during stressful times. Dr. Mahmut Parlak, with his infinite wisdom, always provided a new perspective on any situation, good or bad. Last but not the least, Dhruv, who understood me the most. He counseled me through the rough days and celebrated each of my achievement as his own. A big thank you to all my friends, near and far, who have checked on me regularly and have been involved in this journey with me.

My Ph.D. has been the result of collaborative work with my fantastic lab members and undergraduates. Beilun, Arshdeep, and Dr. Cem Cuscu have helped me in my significant projects. I have also had the opportunity to work with some really talented undergraduates - Byran, Marina, Andrew, Derrick, Eamon, and Chris - during different stages of my projects.

And finally, an acknowledgment of my failures. Behind this work are hundreds of failed attempts and multiple rejection letters. These setbacks have, time and again, strengthened my resolve to keep trying and work harder. As this journey continues, I hope my failures will keep me grounded and motivated to pursue new challenges.

To my parents and my sister for believing in me, always.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Biological sciences are becoming rich in data due to technological advances. It is now possible to obtain information about many living organisms. For example, national biological databases (such as the National Center for Biotechnology Information (NCBI), etc.) store DNA sequence (3 billion-long string) information, activity levels of ∼30,000 genes, cellular profiling data from scores of samples (∼200 cells), and clinical measurements from humans. The growing availability of such information promises a better understanding of important questions (e.g., causes of diseases like cancer). Processing and understanding these "big" data repositories challenge the conventional analysis strategies. This problem has recently interested multiple technology and pharmaceutical companies as its solution can lead to promising advances in the health-care domain. The enormity of the existing data can be gauged from the fact that there are more than 1000 different databases reported so far. The Molecular Biology Database Collection (2007 Update) reported a total of 968 databases, which vary from <100 kB to > 100 GB (e.g. EMBL>500 GB) in size [2] and are growing rapidly. For example, BGI (formerly the Beijing Genomics Institute) generates six terabytes of genomic data every day [3]. Between 100 million to 2 billion human genomes are estimated to be sequenced by the year 2025, far exceeding the growth of big data domains like Astronomy, YouTube, and Twitter [4].

A majority of these biological datasets are sequential in nature. This means that they represent strings (like DNA or protein sequences) or continuous signals that are measurements of activity levels inside the cell. Data-driven approaches like machine learning have shown significant progress in analyzing the existing sequential data. Some of these methods frame the sequential data analysis task as supervised sequence

classification. That is, given a set of labeled data samples, we train our machine learning model to classify the sequences unique to the task. The current state-of-the-art classification methods include (but are not limited to) Instance-based algorithms (Support Vector Machines (SVMs), k-Nearest Neighbor), Ensemble algorithms (Random Forests), and Deep Learning models (Neural Networks). Given the high variability among the biological datasets from different experiments, there is no "one-size-fits-all" solution for all the classification tasks. Therefore, while Deep Neural Networks (DNNs) can handle large datasets accurately and efficiently, they fail to produce accurate results when the size of training samples is small. Furthermore, DNNs are also known as 'black boxes', due to the complex, multi-layer structure, making them difficult to understand (Challenge 1). On the other hand, SVMs, combined with subsequence based string kernel features, are accurate and have theoretical guarantees to converge to a solution. However, they are unable to scale to large datasets due to expanding search space and higher inter-dependencies in the data (Challenge 2).

Dissertation Statement: In this dissertation, we solve two challenges: (1) Interpretability of the predictions for better insights, and (2) Slow computation due to expanding search space of sequential patterns, by improving two popular machine learning models - Deep Neural Networks (DNNs) and String Kernel with Support Vector Machines (SK-SVM).

Challenge 1: Traditional methods for quantifying the relationship between different datasets related to gene expression (surveyed by Dong et al. [5]) suffered a few drawbacks. They either failed to explore large search spaces of possible sequential patterns or were unable to capture dependencies among consecutive subsequence patterns or relied on multiple methods that separated predictions and interpretation analysis. We present a unified discriminative framework using a deep convolutional neural network to classify the sequential datasets. Our model, called DeepChrome [6], allows automatic extraction of complex interactions among important features and is scalable to large datasets. To simultaneously visualize the useful features, we propose a novel optimization-based technique that generates sequential pattern maps from the learned deep model. This visualization provides a general description of underlying mechanisms that regulate genes (`www.deepchrome.org`). DeepChrome outperforms state-of-the-art models like Support Vector Machines and Random Forests for 56 different cell types. The output of our visualization technique not only validates the previous observations but also allows novel insights, some of which have been observed recently in experimental studies.

The visualization technique of DeepChrome gives a general overview of the useful features, but it is unable to provide interpretability at a granular level that can lead to better understanding of the predictions. To further

improve DeepChrome, we add a hierarchical attention based mechanism to learn and highlight important features at different levels of granularity. We call this model, AttentiveChrome. A significant advantage of using the attention mechanism is that we can gain insights and understand the predictions by visualizing 'what' and 'where' the model has focused on while making a prediction. With AttentiveChrome we are able to visualize relevant features for individual genes ($\sim 30,000$) across 56 cell-types.

Challenge 2: Deep Neural Networks (DNNs) provide state-of-the-art performances for various sequence classification problems, analysing DNA [7, 8], and proteins [9, 10]. Notwithstanding their superior performance in accuracy over other traditional approaches, Neural Networks usually require a significant number of training samples. This requirement can be unfeasible for many datasets, especially in the biological research domain. Here, the number of sequences per experiment can be as low as 100. Cost and time constraints may also restrict the size of the training samples.

String kernel (SK) techniques under the support vector machine (SVM) classification framework have provided some of the most accurate results when provided with small labeled datasets (< 5000 sequences) [11, 12, 13, 14, 15, 16]. Through length-k local consecutive substring (k-mer) comparisons that incorporate mismatches and gaps, SK-SVM based models use co-occurrence patterns of local k-mers to calculate the similarity (i.e., so-called kernel function) among sequence samples. This method models the dependencies between consecutive subsequences by using a sliding window of size k. Using the similarity measures, the SK-SVM is trained to classify sequence segments from a set of labeled sequences. Then, the learned models are used to classify a new set of sequences.

k-mer based SK computation can become very slow or even unfeasible when we increase (1) the number of allowed mismatches, and (2) the size of the dictionary, as this drastically expands the sequential pattern search space. This issue is problematic due to several reasons. First, allowing mismatches during substring comparisons is important since biological sequences are prone to mutations, deletions, etc. Second, for robust and scalable estimation of k-mer frequencies, the state-of-the-art SK-SVM method [17] has developed a trie based data structure for computing the kernel matrix among samples. This approach scales exponentially with the dictionary size and the number of mismatches (in the worst case). Therefore, its kernel calculation can be very slow for certain samples. For example, the method takes >5 hours to calculate the kernel matrix for 1 of protein sequence prediction task (dictionary size of protein sequence = 20). Based on this observation, classifying all 500 datasets in the protein database [18] would take ~ 2500 hours. This delay can be a major bottleneck for time-sensitive experiments.

To make the k-mer based SK calculation fast, we present an efficient algorithmic strategy: **Ga**pped **K**ernel with **Co**unting based algorithm, or *GaKCo*. By using a sorted array and counting statistics, GaKCo calculates co-occurrence among k-mers quickly and efficiently and is independent of the dictionary size. As a result, GaKCo derives kernel functions among samples in a fast and memory-efficient manner, making it scalable to large dictionary sizes and a higher number of allowed mismatches. Additionally, GaKCo is naturally parallelizable, and a "multi-thread" variation is implemented to improve the kernel calculation speed. Our results show that the initial GaKCo implementation improves the kernel calculation speed by a factor of 100 for protein classification task over state-of-the-art SK-SVM method [17].

The remainder of the dissertation is organised as follows: Chapter 2 reviews the related work, Chapter 3 describes the DeepChrome model for gene expression prediction, Chapter 4 presents the AttentiveChrome model with focus on interpretability, Chapter 5 discusses GakCo and its comparison with state-of-the-art method, and finally, Chapter 6 lists the contributions of this work and its future directions.

Chapter 2

Literature Review

2.1 Sequential Data in Biology

Sequential data in Biology (or BioSeq data [19]) represents a string (like DNA or protein sequences) or continuous signals that are measurements of activity levels inside the cell. Here, consecutive components of the features have a sequential dependency on each other. Below we describe the different types of sequential data in details:

- *DNA Sequence:* DNA (Deoxyribonucleic Acid) is the building block of the life. It consists of information required by a cell to function properly. Watson-Crick discovered the current-structure of DNA in 1953. The double-helix structure of DNA, consisting of two strands, is made up of four nucleotide bases: Adenine (A), Guanine (G), Thymine (T) and Cytosine(C). 'DNA sequencing' is the process of finding out the precise arrangement of these bases using machines. After sequencing, a DNA sequence looks like a string "ATAAACGACTGAC". A DNA sequences length is measured as 'base-pairs' (bp), where each one base-pair represents one nucleotide in the sequence and its complementary nucleotide on the double helix. Adenine's complementary nucleotide (or base) is Thymine and Guanine's complementary nucleotide (or base) is Cytosine. Thus, knowing the sequence of one strand, one can decipher the sequence of the complementary strand.

Inside each cell, a DNA molecule (3 billion letters bp long) is broken into smaller sections called chromosomes which contain sub-sections called 'genes' that store the regulation information. The 23 pairs of chromosomes consist of about 70,000 genes, and every gene has its function.

- *Protein Sequence:* The coding information of the genes gets converted into proteins via transcription and translation. Proteins are large biomolecules that are involved in almost every process inside a cell ranging from metabolism, cell signaling, immunity, etc. to forming structural and mechanical components of the cell. Just like a DNA is made up of smaller molecules (or nucleotides), a protein is also made up of smaller molecules called 'amino acids'. Therefore, 'protein sequencing' is the practical process of determining the precise order of amino acids in a protein. Unlike DNA sequence, where there are only four characters, a protein sequence can be composed of 20 different characters, each representing an amino acid.

- *Measurements of activity levels along the DNA:* Most of the important events taking place inside the cell, on the DNA, are protein-DNA binding events. For example, Transcription Factors (TFs) are proteins that bind to sequence-specific locations on the DNA near the gene and initiate the process of DNA code conversion to proteins. All these events are captured digitally by using Chromatin immunoprecipitation sequencing, or ChIP-seq. Chromatin immunoprecipitation allows us to separate DNA segments that are involved in the protein binding activities. We then perform DNA sequencing to get the sequences of these DNA segments. Since there are millions of DNA molecules involved in an experiment, we get millions of small DNA sequences from the sequencing machine. These are then aligned to the reference DNA sequence to get an aggregation of reads on the regions of interest. Finally, we count the aggregated reads aligned to the DNA regions, forming smooth sequential signals, and can map and measure the events taking place along the DNA.

Biologists have been collecting the above discussed sequential data for many years now. With next-generation sequencing machines, we can obtain these datasets in large volumes. Therefore, applying data-driven approaches, like machine learning, is a natural way forward to analyze these large repositories. In this dissertation, we solve the challenges associated with this task while focusing on two state-of-the-art machine learning models - Deep Neural Networks (DNNs) and Support Vector Machines (SVMs). We discuss their backgrounds and applications in the following sections.

2.2 Interpretability of Deep Neural Networks

2.2.1 Deep Neural Networks for Biological Tasks

In recent years, the field of biology and medicine has become data-intensive. Under this scenario, deep learning models have become popular in the bioinformatics community, owing to their ability to extract meaningful and hierarchical representations from large datasets.

Neural networks were first proposed in 1943 [20] as a computational explanation for processing of information by the brain. A typical neural network consists of inputs are fed into a hidden layer (or set of functions). This hidden layer then processes the input and then feeds into one or more hidden layers (Figure 2.1). When performing "deep" learning, we use a neural network with multiple hidden layers that eventually produce an output. The training of neural network consists of two stages:

1. *Forward propagation.* During this stage, each layer constructs features, that are refined by subsequent layers.

2. *Backward propagation.* Once the output is produced, the loss (difference from the actual label) is passed, from later layers to previous layers, through the network. Here, each layer adjusts the weights associated with its functions to minimize the loss.

With sufficient data, the neural network can extract or construct features specific to the problem and use them to perform accurate predictions.

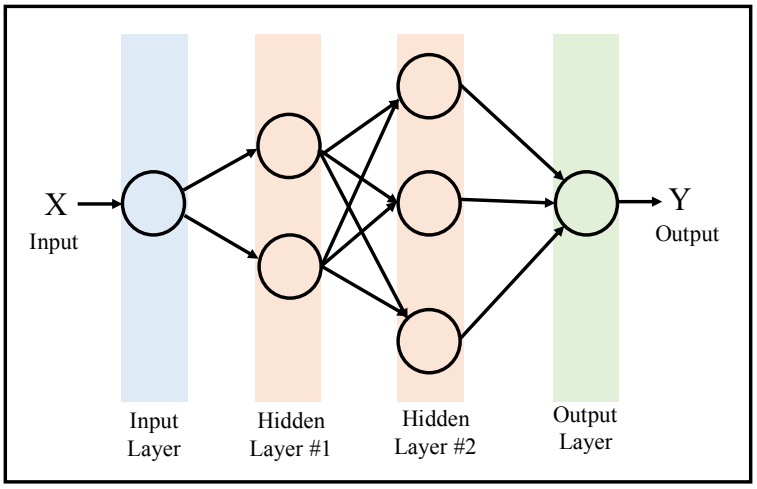

Figure 2.1: Overview of a Deep Neural Network (DNN).

Recently, deep neural networks have led to the groundbreaking performance in many fields such as computer vision [21] and natural language processing [22]. In the biological community too, researchers have been successful in implementing deep neural network models for biological tasks. Qi et al. [9] used a deep multi-layer perceptron (MLP) architecture with multitask learning to perform sequence-based protein structure prediction. Zhou et al. [10] created a generative stochastic network to predict secondary structure on the same data as used by [9]. Recently, Lin et al. [23] outperformed all the state-of-the-art works for protein property prediction task by using a deep convolutional neural network architecture. Leung et al. [24] implemented a deep neural network for predicting alternative splicing patterns in individual tissues and differences of splicing patterns across tissues. Later, Alipanahi et al. [7] applied a convolutional neural network model for predicting sequence specificities of DNA-and RNA-binding proteins as well as generating motifs, or consensus patterns, from the features that were learnt by their model. Lanchantin et al. [25] proposed a deep convolutional/highway MLP framework for the same task and demonstrated improved performance. Similarly, Zhou et al. [26] used DNA sequences as inputs to predict different chromatin features and understand the effect of non-coding variants on these measurements of interest.

Despite the success, the field of biology poses some unique challenges for the machine learning or deep learning community. For example, neural networks usually require a significant number of training samples for good performance. However, the number of samples for biological experiments can be meager and thus, making it unfeasible to implement neural networks on these datasets. Also, while deep learning models have proven to be very accurate, they have widely been viewed as "black boxes". Although this may not be an issue for text or image classification cases, in biology, interpretability of the model is a desirable trait. Biologists are not only interested in predicting a particular outcome but also require explainability of a specific prediction to understand the underlying mechanism.

Researchers have attempted to develop separate visualization techniques that explain a deep classifier's decisions. Most prior studies have focused on understanding Convolutional Neural Networks (CNN) for image classifications. Following Shrikumar et al. [27], we roughly categorize these studies into the following groups:

1. *Perturbation-based methods.* These methods assign importance to input features by making perturbations to individual inputs and observing their impact on later layers of the network. For example, Zeiler et al. [28] covered up different parts of an input image and visualized the changes in the outputs of later layers of their neural network. This strategy was adopted by Zhou et al. [26] to quantify the

impact of virtual mutations at different positions on the DNA. Instead of studying perturbation for each input point, Zintgraf et al. [29] marginalized these differences over input patches. One of the major drawbacks of these methods is that they can be computationally expensive as one has to repeat the training step after each perturbation.

2. *Deconvolution methods.* The "deconvolution' approach [28] maps hidden layer representations back to the input space for a specific example, showing those features of an image that are important for classification. Alipanahi et al. [7] used this method to generate consensus sequential patterns (or motifs) of DNA sequences that had a protein-binding site.

3. *Gradient-based methods.* "Saliency maps" are generated by using the gradient of the output with respect to the input. This gradient is calculated using a backpropagation pass such that the importance signal from the output layer is passed towards the input. Multiple studies [30, 31, 32] use a first-order Taylor expansion to linearly approximate the deep network and seek most relevant input features. Bach et al. [33] introduced the concept of Layerwise Relevance Propagation (LRP) importance scores. These scores were roughly equivalent to the element-wise product of saliency map outputs and the inputs. This formulation leverages the sign and strength of the input signal. Gradient-based methods present their own set of challenges for interpretability. One such example is that when working with neural networks with nonlinear activation layers (or functions), the gradients zeroed out during backpropagation can fail to highlight important inputs that contribute negatively to output.

4. *Optimization-based methods.* The "class optimization" based visualization [30] tries to find the best example (through optimization of a randomly generated input) that maximizes the probability of the class of interest. Lanchantin et al. [25] has compared both salience map based as well as optimization based methods for generating motifs for DNA-protein binding sites.

5. *Difference-to-reference methods.* These methods define a "reference" input and aim to explain the difference of the given input with respect to the reference. Sundararajan et al. [34] use the integration of gradients while scaling some reference to the input values. Shrikumar et al. [27] use a similar idea but instead of integrating the gradient, they multiply and propagate absolute contribution of differences with respect to the change in input. These strategies, while effective, require domain knowledge to define a "reference" input to study the differences.

Some recent studies [35, 36] explored the interpretability of recurrent neural networks (RNN) for text-based tasks.

2.2.2 Attention-based Deep Neural Networks

The idea of attention in deep learning arises from the properties of the human visual system. When perceiving a scene, the human vision gives more importance to some areas over others [37]. This adaptation of "attention" allows deep learning models to focus selectively on only the important features. Deep neural networks augmented with attention mechanisms have obtained great success on multiple research topics such as machine translation [38], object recognition [39, 40], image caption generation [41], question answering [22], text document classification [42], video description generation[43], visual question answering -[44], or solving discrete optimization [45]. Attention brings in two benefits: (1) By selectively focusing on parts of the input during prediction the attention mechanisms can reduce the amount of computation and the number of parameters associated with deep learning model [39, 40]. (2) Attention-based modeling allows for learning salient features dynamically as needed [42], which can help improve accuracy.

Different attention mechanisms have been proposed in the literature, including 'soft' attention [38], 'hard' attention [41, 46], or 'location-aware' [47]. Soft attention [38] calculates a 'soft' weighting scheme over all the component feature vectors of input. These weights are then used to compute a weighted combination of the candidate feature vectors. The magnitude of an attention weight correlates highly with the degree of significance of the corresponding component feature vector to the prediction. Inspired by Yang et al. [42], AttentiveChrome uses two levels of soft attention for predicting gene expression from HM marks.

Moreover, since attention in models allows for automatically extracting salient features, attention-coupled neural networks impart a degree of interpretability. By visualizing what the model attends to, attention can help gauge the predictive importance of a feature and hence interpret the output of a deep neural network [42].

2.3 Sequence Classification with String Kernels

2.3.1 Support Vector Machine (SVM):

When number of training sequence samples are small (< 5000 sequences) string kernel (SK) techniques under the support vector machine (SVM) classification framework have provided some of the most accurate results [11, 12, 13, 14, 15, 16]. Support Vector Machines is a popular learning method used for binary classification. Introduced by Vladimir N. Vapnik [48], it finds a hyperplane which separates a d-dimensional feature space into two classes.

Given N number of total training samples with inputs and outputs: $\{x_i, yi\}$ such that $i = 1, \ldots, N$ and $x_i \in R^d$, where R^d is a d-dimensional feature space. Also, $y_i \in \{1, 1\}$ that is the output is a binary class label. Thus, all the hyperplanes in the R^d feature space can be expressed as:

$$w.x + b = 0 \tag{2.1}$$

here, w is a parameter and b is a constant.

If the above hyperplane separates the data, we can define a decision function:

$$f(x) = sign(wx + b) \tag{2.2}$$

that can classify the training data accurately. However, since there can be multiple hyperplanes satisfying the above equation, the concept of "margin" is introduced. That is we define canonical hyperplanes that separate the nearest points of the data from the separating hyperplane by a distance of at least 1. They are formulated as:

$$x_i w + b \geq +1, y_i = +1 \tag{2.3}$$

$$x_i w + b \leq -1, y_i = -1 \tag{2.4}$$

Or more concisely,

$$y_i(x_i w + b) \geq 1, \forall i \tag{2.5}$$

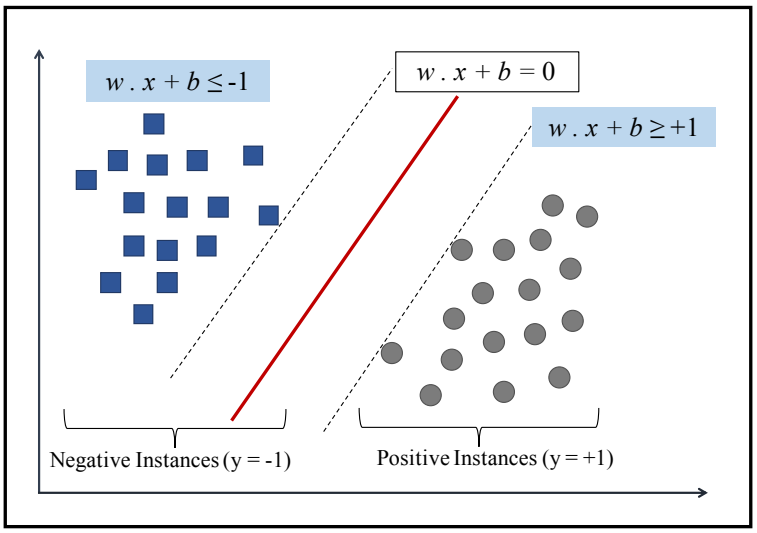

Figure 2.2: Overview of a Support Vector Machine (SVM).

Among all possible hyperplanes, SVM selects the one where the distance between the canonical hyperplanes or the "margin" is as large as possible (Figure 2.2). That is, we want the hyperplane that maximizes the geometric distance to the closest data points. Solving and subtracting the two distances we get the summed distance from the separating hyperplane to the canonical hyperplanes as our maximum margin that is, $\frac{2}{||w||}$. Therefore, finding the optimum separating hyperplane can be formulated as a quadratic optimization problem, which solves for w and b. Constructing this problem as a dual problem with Lagrangian multiplier α_i gives us the final classification function:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i x_i . x + b \tag{2.6}$$

It is worth noting here that the VC-dimension (a measure of a systems likelihood to perform well on unseen data) of SVMs can be explicitly calculated. This property makes them theoretically well-founded in contrast to other learning methods like DNNs. SVMs have also been used to solve regression tasks. In regression task, we train our model to output a numerical value instead of assigning classes.

2.3.2 String Kernels

The key idea of string kernels is to apply a function $\phi(\cdot)$, which maps strings of arbitrary length into a vectorial feature space of fixed length. In this space, we apply a standard classifier such as SVM [48]. In the real-world scenario, most of the data spaces are not linearly separable and therefore, the notion of a "kernel induced feature space" was introduced, which casts the data into a higher dimensional space where the data is separable. Kernel-version of SVMs calculates the decision function for an input sample x as:

$$f(x) = \sum_{i=1}^{N} \alpha_i y_i K(x_i, x) + b \tag{2.7}$$

where N is the total number of training samples. String kernels [11, 12, 17], implicitly compute $K(x, x')$ as an inner product in the mapped feature space $\phi(x)$ as:

$$K(x, x') = \langle \phi(x), \phi(x') \rangle, \tag{2.8}$$

where $x = (s_1, \ldots, s_{|x|})$. $x, x' \in \mathcal{S}$. $|x|$ denotes the length of the string x. $\mathcal{S}$ represents the set of all strings composed of dictionary Σ. $\phi : \mathcal{S} \rightarrow R^p$ defines the mapping from a sequence $x \in \mathcal{S}$ to a p-dimensional feature vector.

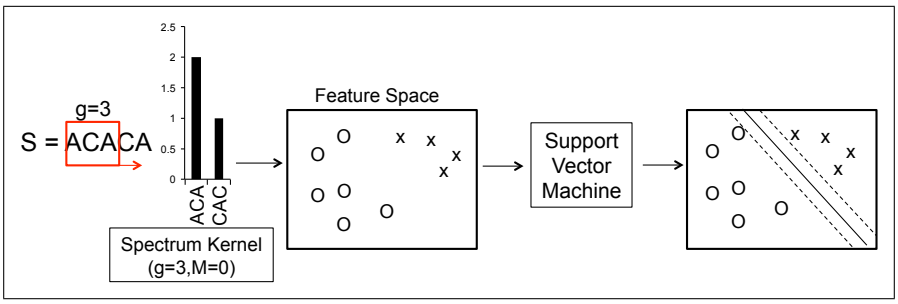

Figure 2.3: Overview of the String Kernel + SVM model.

The feature representation $\phi(\cdot)$ plays a vital role in the effectiveness of string analysis since it is hard to describe strings as feature vectors. One classical method to represent a string is as an unordered set of k-mers, or combinations of k adjacent characters. A feature vector indexed by all k-mers records the number of occurrences of each k-mer in the current string. The string kernel using this representation is called spectrum kernel [49] (see Figure 2.3), where the spectrum representation counts the occurrences of each k-mer in a string. Kernel scores between strings are computed by taking an inner product between corresponding "k-mer

- indexed" feature vectors:

$$K(x, x') = \sum_{\gamma \in \Gamma_k} c_x(\gamma) \cdot c_{x'}(\gamma) \tag{2.9}$$

where γ represents a k-mer, Γ_k is the set of all possible k-mers, and $c_x(\gamma)$ is the number of occurrences (with normalization) of k-mer γ in string x.

A a few other notable string kernels include (but are not limited to):

1. (k, m)-*Mismatch Kernel.* This kernel calculates the dot product of contiguous k-mer counts with m mismatches allowed [11, 12].

2. (g, k)-*Gappy Kernel.* For this kernel [11], the feature map is calculated by counting gappy matches of g-mers to k-mer features (here, $g \geq k$). Thus, given a g-mer of length g, the kernel uses the contiguous k-mer counts, allowing upto $(g - k)$ gaps.

3. *Weighted Degree Kernel.* This kernel [50] incorporates positional information of the k-mers. It counts the exact co-occurrences of k-mers at corresponding positions in the two sequences and these counts are weighted by a coefficient that is proportional to the length of the matching k-mers. This weighting scheme assigns higher kernel scores to longer k-mer matches.

4. *Substring Kernel.* It measures the similarity between sequences based on common co-occurrence of exact matching subpatterns (e.g., substrings) [51].

5. *Profile Kernel.* This method uses the notion of similarity based on a probabilistic model (e.g., profile) [52].

6. *Cluster Kernel.* The "sequence neighborhood" kernel or "cluster" kernel [53] is a semi-supervised extension of the string kernel. It replaces every sequence with a set of "similar" (neighboring) sequences and obtains a new representation. Then, it averages over the representations of these contiguous sequences found in the unlabeled data using a sequence similarity measure.

All string kernels perform string matching by calculating the feature representation $\phi(\cdot)$ using the counts of k-mer occurrence. Thus, the following sections briefly discuss the notable methods for string matching and counting the occurrence of k-mers (mostly in the bioinformatics literature).

2.3.3 String Matching Algorithms

String or pattern matching algorithms are an important category of string-related algorithms in computer science. The dictionary of a given language, Σ, is a finite set of symbols. String or pattern matching task matches an entity, that is a vector of the elements of Σ. In bioinformatics, this vector is usually a k-mer, for example "ATATCG" is a k-mer (k=6) from the dictionary $\Sigma = \{A, T, C, G\}$. In natural language processing, this entity can be words or k-mers formed from characters. The simplest algorithm is Naive String Search, which matches the characters of the query pattern and text at each position. This method is very computationally costly for a large number of texts or patterns.

Approximate string further complicates the task by allowing three types of changes in the text/pattern: (1) insertion, (2) deletion, and (3) substitution of characters. These modifications are necessary for biological sequences, which are prone to mutations and exact matching algorithms were not able to provide the desired accuracy. [1] Under this scenario, finite state automaton based matching was introduced by Ukkonen et al. [55]. Here, deterministic finite automaton (DFA) were constructed out of the dictionary Σ and used to find a stored pattern. This method was quick to locate the string, but it was costly to build the DFA.

Later, index-based strategies like suffix trees and suffix arrays became very popular. Blumer et al. [56] refined the suffix trees from earlier implementations of trie data structures (used in multiple string kernels). Chang et al. [57] proposed suffix trees for approximate string matching in biological applications like overlap detection for DNA sequence assembly. Manber et al. [58] presented a suffix array (sorted list of suffixes of a string) as an improvement over suffix trees. They claimed that despite longer construction times, suffix arrays were space efficient with better pattern searching times than suffix trees. Recently, multiple suffix array compression methods, like Burrows-Wheeler Transform with FM-Index [59, 60], have been used to speed up string matching applications where the size of the text is enormous (assembly of genomes, i.e., 3 billion length DNA sequence). Most of the current tools for such applications consist of a k-mer counting step where the matching k-mers are calculated to filter out unique DNA fragments to reduce feature space. Next section discusses the state-of-the-art methods for k-mer counting.

[1] The concept of *edit distance* was introduced to measure the similarity among strings with different modifications (insertions, deletions or substitutions). *Hamming distance* (used to measure string similarity for string kernels) is a simplified form of edit distance that only measures the degree of substitutions in the strings [54].

2.3.4 k-mer counting methods

k-mer counting is the method by which we determine the number of matching or unique k-mers in any text or pattern. Tools handling large text datasets need to filter out these unique k-mers to reduce the processing or counting time. GaKCo uses a 'sort and count' method for calculating the number of matching substrings to compute the mismatch profile. This is a widely used method that lists all the substrings, sorts them lexicographically and counts all the consecutive matching entries while skipping the unique ones. It has been used previously in tools used for genome assembly [61], discovery of motifs (or most common fixed length patterns) [62], and string kernel calculation [12].

Chapter 3

Deep Neural Network for Gene Expression Prediction

3.1 Predicting Gene Expression using Histone Modifications

Gene regulation is the process of how the cell controls which genes are turned "on" (expressed) or "off" (not-expressed) in its genome. The human body contains hundreds of different cell types, from liver cells to blood cells to neurons. Although these cells include the same set of DNA information, their functions are different [1]. The regulation of different genes controls the destiny and function of each cell. In addition to DNA sequence information, many factors, especially those in its environment (i.e., chromatin), can affect which genes the cell expresses. This chapter proposes a deep learning architecture to learn from data how different chromatin factors influence gene expression in a cell. Such understanding of gene regulation can enable new insights into principles of life, the study of diseases, and drug development.

"Chromatin" denotes DNA and its organizing proteins [2]. A cell uses specialized proteins to organize DNA in a condensed structure. These proteins include histones, which form "bead"-like structures that DNA wraps around, in turn organizing and compressing the DNA. An important aspect of histone proteins is that they are prone to chemical modifications that can change the spatial arrangement of DNA. These spatial

[1] DNA is a long string of paired chemical molecules or nucleotides that fall into four different types and are denoted as A, T, C, and G. DNA carries information organized into units such as genes. The set of genetic material of DNA in a cell is called its genome.

[2] The complex of DNA, histones, and other structural proteins is called chromatin.

re-arrangements result in certain DNA regions becoming accessible or restricted and therefore affecting expressions of genes in the neighborhood region. Researchers have established the "Histone Code Hypothesis" that explores the role of histone modifications in controlling gene regulation. Unlike genetic mutations, chromatin changes such as histone modifications are potentially reversible ([63]). This crucial difference makes the understanding of how chromatin factors determine gene regulation even more impactful because this knowledge can help developing drugs targeting genetic diseases.

At the whole genome level, researchers are trying to chart the locations and intensities of all the chemical modifications, referred to as marks, over the chromatin [3]. Recent advances in next-generation sequencing have allowed biologists to profile a significant amount of gene expression and chromatin patterns as signals (or read counts) across many cell types covering the full human genome. These datasets have been made available through large-scale repositories, the latest being the Roadmap Epigenome Project (REMC, publicly available) ([1]). REMC recently released 2,804 genome-wide datasets, among which 166 datasets are gene expression reads (RNA-Seq datasets) and the rest are signal reads of various chromatin marks across 100 different "normal" human cells/tissues [1].

The fundamental aim of processing and understanding this repository of "big" data is to understand gene regulation. For each cell type, we want to know which chromatin marks are the most important and how they work together in controlling gene expression. Computational tools should consider two important properties when modeling such data.

- First, signal reads for each mark are spatially structured and high-dimensional. For instance, to quantify the influence of a histone modification mark, learning methods typically need to use as input features all of the signals covering a DNA region of length $10,000$ base pair (bp) [4] centered at the transcription start site (TSS) of each gene. These signals are sequentially ordered along the genome direction. To develop "epigenetic" drugs, it is important to recognize how a chromatin mark's effect on regulation varies over different genomic locations.

- Second, various types of marks exist in human chromatin that can influence gene regulation. For example, each of the five standard histone proteins can be simultaneously modified at multiple different sites with various kinds of chemical modifications, resulting in a large number of different histone modification marks. For each mark, a feature vector is created, representing its signals surrounding a

[3] In biology this field is called epigenetics. "Epi" in Greek means over. The epigenome in a cell is the set of chemical modifications over the chromatin that alter gene expression.

[4] A base pair refers to one of the double-stranded DNA sequence characters (ACGT)

gene's TSS position. When modeling genome-wide signal reads from multiple marks, learning algorithms should take into account the modular nature of such feature inputs, where each mark functions as a module. We want to understand how the interactions among these modules influence the prediction (gene expression).

3.2 Previous Computational Methods

Computational methods have shown initial success in modeling and understanding interactions among chromatin features, such as histone modification marks, to predict gene expression. Initial studies, like [64] and [65], investigated experimentally the correlation between histone modification marks and gene regulation. Karlic et al. [66] established that there exists a quantitative relationship between histone modifications and gene expression. They applied a linear regression model on histone modification signals and predicted gene expression from human T-cell studies ([67]). They reported a high correlation of their predictions with the observed gene expressions (Pearson coefficient $r = 0.77$) and showed that a combination of only two to three specific modifications is sufficient for making accurate predictions. Extending this concept further, Costa et al. [68] implemented a mixture of several linear regression models to extract the relative importance of each histone modification signal and its effect on gene expression (high or low). This study confirmed the activator and repressor roles of H3K4me3 and H3K27me3 respectively. It also demonstrated that a mixture of two regression models performs better than a single regression model. Both these studies applied relatively simple modeling on a small dataset. They used the mean signal of the whole transcription start site (TSS) flanking regions as input features. This leads to a potential bias since histone modification signals exhibit diverse patterns of local distributions with regard to different genes. Ignoring the details of these neighborhood patterns is undesirable.

Cheng et al. [69] applied Support Vector Machine (SVM) models on worm datasets ([70]) and reformulated the task as gene expression classification and prediction. The authors divided regions flanking transcription start site (TSS) and transcription termination site (TTS) into 100 base-pair (bp) bins and used the histone modification signal in each bin as a feature for the SVM. To incorporate information from all positions or bins, they trained different models for different bins that resulted in 160 models for 160 bins. They validated the existence of the quantitive relationship between histone modifications and gene expression by such bin-specific modeling. Furthermore, using a separate linear regression model, the paper inferred pair-wise interactions

between different histone modifications using binary combinatorial terms. Since it is infeasible to consider all possible higher order interaction terms through polynomial regression, Bayesian networks were then used for modeling such relationships. However, Bayesian networks do not take into consideration local neighboring bin information and their highly connected output network is difficult to interpret.

Using a similar experimental setup, Dong et al. [71], applied a Random Forest Classifier on histone modification signals to classify gene expression as high or low. They then used the classified outputs as inputs to a linear regression model to predict the gene expression value. They used human datasets across 7 different cell types ([72]) and reported a high correlation (Pearson coefficient $r = 0.83$) between predicted and actual gene expressions. To include information from all bins into a single model, the authors performed feature selection where only the bin value which correlated the most with gene expression was used as input. For combinatorial analysis, instead of studying all possible combinations, the 11 histone modifications were grouped into four functional categories. These groupings were used to determine prediction accuracy based on each category as a sole feature as well as combinations of different categories. This technique gives a broader picture of the combinatorial effect. However, individual details of histone modifications are missed. In addition, this approach cannot capture the possible influence of other bins besides the "best bin" for gene regulation.

In order to to elucidate the possible combinatorial roles of histone modifications in gene regulation, Ho et al. [73] applied rule learning on the T-cells datasets ([67]) and produced 83 valid rules for gene expression (high) and repression (low). The authors selected the 20 most discriminative histone modifications as input into a rule learning system. They used several heuristics to filter out unexpected rules that were obtained by the learning system after scanning the entire search space. However, this study does not consider detailed feature patterns across local bins and does not perform prediction of gene expression.

Ernst et al. [74] leveraged the correlated nature of epigenetic signals in the REMC database, including histone modifications. Their tool, ChromImpute, imputed signals for a particular new sample using an ensemble of regression trees on all the other signals and samples. EFilter ([75]), a linear estimation algorithm, predicted gene expression in a new sample by using imputed expression levels from similar samples. Unlike the studies discussed above, these works focus on imputing or predicting signals for new samples.

Table 3.1: Comparison of previous studies for the task of quantifying gene expression using histone modification data.

The columns indicate properties (a) whether the study has a unified end-to-end architecture or not (b) if it captures non-linearity among features (c) how has the bin information been incorporated (c) if representation of features is modeled on local and global scales (d) whether gene expression prediction is provided and finally, (e) if combinatorial interactions among histone modifications are modeled. DeepChrome is the only model that exhibits all six desirable properties.

Computational Study	Unified Strategy	Non-linear model	Including Bin Info	Representation Learning		Prediction	Combinatorial Interactions
				Neighboring bins	Whole Region		
Linear Regression ([66])	×	×	×	×	✓	✓	×
Support Vector Machine ([69])	×	✓	Bin-specific strategy	×	✓	✓	✓
Random Forest ([71])	×	✓	Best-bin strategy	×	✓	✓	×
Rule Learning ([73])	×	✓	×	×	✓	×	✓
DeepChrome	✓	✓	Automatic	✓	✓	✓	✓

3.3 Approach

Previous computational methods failed to capture higher-order combinatorial effects among histone modifications, used bin related strategies that cannot represent neighboring bins, or relied on multiple methods to separate prediction and combinatorial analysis. We utilize a deep convolutional neural network model for predicting gene expression from histone modification data. The network automatically learns both the combinatorial interactions and the classifier jointly in one unified discriminative framework, eliminating the need for human effort in feature engineering. Since the combinatorial effects are automatically learned through multiple layers of features, we present a visualization technique to extract those interactions and make the model interpretable.

3.3.1 Input Generation

Aiming to systematically understand the relationship between gene regulation and histone modifications, we divided the $10,000$ basepair (bp) DNA region ($+/-5000$ bp) around the transcription start site (TSS) of each gene into bins of length 100 bp. Each bin includes 100 bp long adjacent positions flanking the TSS of a gene. In total, we consider five core histone modification marks from REMC database ([1]), which are summarized in Table 3.2. These five histone modifications are selected as they are uniformly profiled across all cell-types considered in this study. They include include (we rename these HMs in our analysis for readability): H3K27me3 as H_{reprA}, H3K36me3 as H_{struct}, H3K4me1 as H_{enhc}, H3K4me3 as H_{prom}, and H3K9me3 as

H_{reprB}. HMs H_{reprA} and H_{reprB} are known to repress the gene expression, H_{prom} activates gene expression, H_{struct} is found over the gene body, and H_{enhc} sometimes helps in activating gene expression. This makes the input for each gene a 5×100 matrix, where columns represent different bins and rows represent histone modifications. For each bin, we report the value of all 5 histone signals as the input features for that bin (Figure 3.1). We formulate the gene expression prediction as a binary classification task. Specifically, the outputs of DeepChrome are labels $+1$ and -1, representing gene expression level as high or low, respectively. Following [69], we use the median gene expression across all genes for a particular cell-type as a threshold to discretize the gene expression target. Figure 3.1 summarizes our input matrix generation strategy.

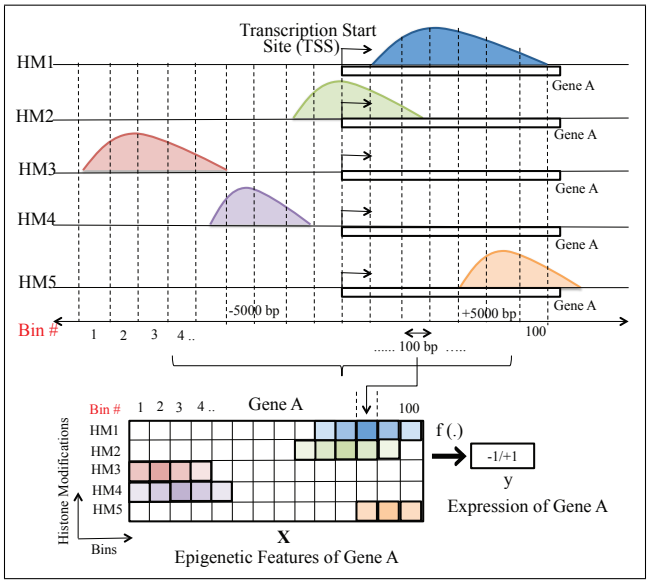

Figure 3.1: Feature Generation for DeepChrome model.

Bins of length 100 base-pairs (bp) are selected from regions $(+/-5000$ bp) flanking the transcription start site (TSS) of each gene. The signal value of all five selected histone modifications in bins forms input matrix X, while discretized gene expression (label $+1/-1$) is the output y.

Our setup is similar to [69] and [71], except that we primarily focus on the regions around TSS instead of also including regions from gene body or transcription termination site (TTS). This is based on the observations from [69] showing that signals close to the TSS are the most informative, therefore eliminating the need to obtain bins from regions toward the end of the gene. In addition, due to the scalability of CNNs, we were able to use larger regions flanking TSS than previous studies in order to better capture effects of distal signals as well as to cover more regions. This therefore enhances the possibility to model long range interactions among histone modifications.

Table 3.2: Five core histone modification marks, as defined by REMC study [1], along with their functional categories

Histone Mark	Associated with	Renamed as	Functional Category
H3K4me3	Promoter regions	H_{prom}	Promoter mark
H3K4me1	Enhancer regions	H_{enhc}	Distal mark
H3K36me3	Transcribed regions	H_{struct}	Structural mark
H3K27me3	Polycomb repression	H_{repA}	Repressor mark
H3K9me3	Heterochromatin regions	H_{repB}	Repressor mark

3.3.2 An end-to-end architecture based on Convolutional Neural Network (CNN)

Convolution Neural Networks (CNNs) were first popularized by LeCun et al. [76] and have since been extensively used for a wide variety of applications. We have implemented a CNN for gene expression classification task using the Torch7 ([77]) framework. Our DeepChrome model, summarized in Figure 3.2, is composed of five stages. We assume our training set contains N_{samp} gene samples of the labeled-pair form $(\mathbf{X}^{(n)}, y^{(n)})$, where $\mathbf{X}^{(n)}$ are matrices of size $N_f(=5) \times b \ (= 100)$ and $y^{(n)} \in \{-1, +1\}$ for $n \in \{1, ..., N_{samp}\}$.

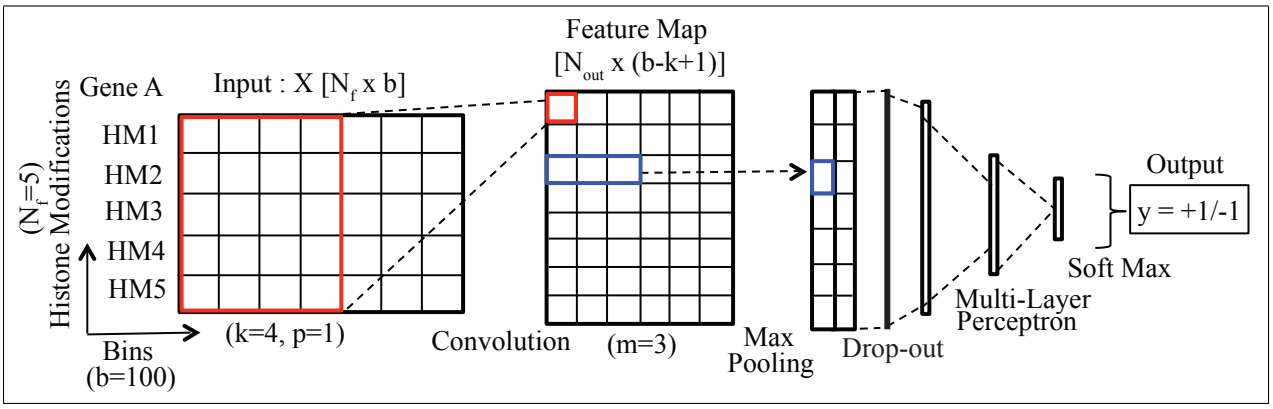

Figure 3.2: DeepChrome convolution neural network (CNN) model

The input matrix X, comprising of 100 bins with signals from five histone modifications, goes through different CNN stages. These stages are : convolution, pooling followed by dropout, and multi-layer perceptron with alternating linear and non-linear layers. Softmax function, in the end, maps the output from the model into classification prediction.

1. **Convolution:** We use temporal convolution with N_{out} filters, each of length k. This performs a sliding window operation across all bin positions, which produces an output feature map of size $N_{out} \times (b-k+1)$. Each sliding window operation applies N_{out} different linear filters on k consecutive input bins from position $p = 1$ to $(b - k + 1)$. In Figure 3.2, the red rectangle shows a sliding window operation with $k = 4$ and $p = 1$. Given an input sample $\mathbf{X}$ of size $N_f \times b$, the feature map, $\mathbf{Z}$, from convolution is computed as follows :

$$\mathbf{Z} = f_{conv}(\mathbf{X})$$
$$\mathbf{Z}_{p,i} = \mathbf{B}_i + \sum_{j=1}^{N_f} \sum_{r=1}^{k} \mathbf{W}_{i,j,r} \mathbf{X}_{p+r-1,j} \tag{3.1}$$

This is generated for the p^{th} sliding neighborhood window and the i^{th} hidden filter, where $p \in \{1, ..., (b-k+1)\}$ and $i \in \{1, ..., N_{out}\}$. $\mathbf{W}$, of size $N_{out} \times N_f \times k$, and $\mathbf{B}$, of size $N_{out} \times 1$, are the trainable parameters of the convolution layer and N_{out} denotes the number of filters.

2. **Rectification:** In this stage, we apply a non-linearity function called rectified linear unit (ReLU). The ReLU is an element-wise operation that clamps all negative values to zero:

$$f_{relu}(z) = \text{relu}(z) = \max(0, z) \tag{3.2}$$

3. **Pooling:** Next, in order to learn translational invariant features, we use temporal maxpooling on the output from the first two steps. Maxpooling simply selects the max values in a certain range, which forms a smaller representation of a large TSS-proximal region for a given gene. Maxpooling is applied on an input $\mathbf{Z}$ of size $N_{out} \times P$, where $P = (b-k+1)$. With a pooling size of m, we obtain an output $\mathbf{V}$ of size $N_{out} \times \lfloor \frac{P}{m} \rfloor$:

$$\mathbf{V} = f_{maxpool}(\mathbf{Z})$$
$$\mathbf{V}_{i,p} = \max_{j=1}^{m} \mathbf{Z}_{i,m(p-1)+j} \tag{3.3}$$

where $p \in \{1, ..., \lfloor \frac{P}{m} \rfloor\}$ and $i \in \{1, ..., N_{out}\}$. In Figure 3.2, the blue rectangle shows the result of a maxpooling operation on the feature map where $m = 3$.

4. **Dropout:** The output is then passed though a dropout layer ([78]), which randomly zeroes the inputs to the next layer during training with a chosen probability of 0.5. This regularizes the network and prevents over-fitting. It resembles ensemble techniques, like bagging or model averaging, which are very popular in bioinformatics.

5. **Classical feed-forward neural network layers:** Next, the learnt region representation is fed into a multi-layer perceptron (MLP) classifier to learn a classification function mapping to gene expression labels. This standard and fully connected multi-layer perceptron network has multiple alternating

linear and non-linear layers. Each layer learns to map its input to a hidden feature space, and the last output layer learns the mapping from a hidden space to the output class label space $(+1/-1)$ through a softmax function.

Figure 3.2, shows a MLP with 2 hidden layers and a softmax function at the end. This stage is represented as $f_{mlp}(.)$.

The whole network output form can be written as:

$$f(\mathbf{X}^{(n)}) = f_{mlp}(f_{maxpool}(f_{relu}(f_{conv}(\mathbf{X}^{(n)})))) \tag{3.4}$$

All the above stages are effective techniques that are widely practiced in the field of deep learning. All parameters, denoted as Θ, are learned during training in order to minimize a loss function which captures the difference between true labels y and predicted scores from $f(.)$.[5] The loss function L, on the entire training set of size n, is defined:

$$L = \sum_{n=1}^{N_{samp}} loss(f(\mathbf{X}^{(n)}), y^{(n)}) \tag{3.5}$$

We use stochastic gradient descent (SGD) ([79]) to train our model via backpropagation. For a set of training samples, instead of calculating the true gradient of the objective using all training samples, SGD calculates the gradient per sample and updates accordingly on each training sample. For our objective function, the loss $L(.)$ (equation 3.5) is minimized by the gradient descent step that is applied to update network parameters Θ as follows:

$$\Theta \leftarrow \Theta - \eta \frac{\partial L}{\partial \Theta} \tag{3.6}$$

where η is the learning rate (set to 0.001).

3.3.3 Visualizing combinatorial effect through optimization

In addition to being able to make high accuracy predictions on the gene expression task, an important contribution of DeepChrome is that it allows us to discover and visualize the combinatorial relationships between different histone modifications which lead to such predictions. Until recently, deep neural networks

[5]When training this deep model, parameters, at first, are randomly initialized and input samples are fed through the network. The output of this network is a score prediction associated with a sample. The difference between a prediction output $f(\mathbf{X})$ and its true label y is fed back into the network through a 'back-propagation' step.

were viewed as "black boxes" due to the automatically learned features spanning multiple layers. Since gene expression is dependent on the combinatorial interactions among histone modifications, it is critical to understand how the network extracts features and makes its predictions. In other words, we wish to understand the combinatorial patterns of histone modifications which lead to either a high or low gene expression prediction by the network. We attempt to do this by extracting a map of feature patterns that are highly influential in predicting gene expression directly from the trained network. This approach, called a network-centric approach ([80]), finds the *class specific* features from the trained model and is independent of specific testing samples.

The technique we use to generate this visualization was inspired from works by Simonyan et al. [30] and Yosinski et al. [80], which seek to understand how a convolutional neural network interprets a certain image class on the task of object detection. We, instead, seek to find how our network interprets a gene expression class (high or low). Given a trained CNN model and a label of interest (+1 or −1) in our case, we perform a numerical optimization procedure on the model to generate a feature pattern map which best represents the given class. This optimization includes four major steps:

1. Randomly initialize an input $\mathbf{X}_c$ (of size $N_f(=5) \times b(=100)$).

2. Find the best values of entries in $\mathbf{X}_c$ by optimizing the following equation(3.7). We search for $\mathbf{X}_c$ so that the loss function is minimized with respect to the desired labels +1 (gene expression = high) or −1 (gene expression = low). Using equation (3.4), $f(\mathbf{X}_c)$ provides the predicted label using the trained DeepChrome model on an input $\mathbf{X}_c$. We would like to find an optimal feature pattern, $\mathbf{X}_c$, so that its predicted label $f(\mathbf{X}_c)$ is close to the desired class label c:

$$\arg\min_{X_c} L_{visual} = \arg\min_{X_c} \{L(f(\mathbf{X}_c), y = c) + \lambda \|\mathbf{X}_c\|_2^2\} \tag{3.7}$$

   where $c = +1$ or -1, $L(.)$ is the loss function defined in equation (3.5). L_2 regularization, $\|\mathbf{X}_c\|_2^2$, is applied to scale the signal values in $\mathbf{X}_c$, and λ is the regularization parameter. A locally-optimal X_c can be found by the back-propagation method. This step is similar to the CNN training procedure, where back-propagation is used to minimize the loss function by optimizing the *network* parameters Θ. However, in this case, the optimization is performed with respect to the *input values* ($\mathbf{X}_c$) and the network parameters are fixed to the values obtained from the classification training. $\mathbf{X}_c$ is optimized in

the following manner:

$$\mathbf{X}_c^{t+1} \leftarrow \mathbf{X}_c^t - \alpha \frac{\partial L_{visual}}{\partial \mathbf{X}_c} \tag{3.8}$$

where α is the learning rate parameter and t represents the iteration step of the optimization.

3. Next, we set all the negative output values to 0 and normalize $\mathbf{X}_c \in [0, 1]$:

$$\mathbf{X}_{c(norm)} = \frac{\mathbf{X}_c}{max(\mathbf{X}_c)} \tag{3.9}$$

4. Finally, we set a threshold of 0.25 to define "active" bins. Bins in $\mathbf{X}_{c(norm)}$ with values > 0.25 are considered important since they indicate that such histone modification signals are important for predicting particular class. We count the frequency of these active bins for a particular histone modification mark. A high frequency count ($>$ mean frequency count across all histone modification marks) of active bins indicates the important influence of these histone modification signals on target gene expression level.

This visualization technique represents the notion of a class that is learnt by the DeepChrome model and is not specific to a particular gene. The optimized feature pattern map $\mathbf{X}_{c(norm)}$ is representative for a particular gene expression label of $+1$ (high) or -1 (low). In Figure 3.5, DeepChrome visualizes $\mathbf{X}_{c(norm)}$ as heat-maps. Through these maps, we obtain intuitive outputs for understanding the combinatorial effects of histone modifications on gene regulation.

3.4 Experimental Setup

3.4.1 Dataset

We downloaded gene expression levels and signal data for five core histone modification signals for 56 different cell types from the REMC database ([1]). REMC is a public resource of human epigenomic data produced from hundreds of cell-types. Core histone modification marks, as defined by the REMC study [1], have been listed in Table 3.2 and are known to play important roles in gene regulation. We focus on these "core" histone modifications as they have been uniformly profiled for all 56 cell types through sequencing technologies. The gene expression data has been quantified for all annotated genes in the human genome and has been

normalized for all 56 cell types in the REMC database. As mentioned before, the target problem has been formulated as a binary classification task. Thus, each gene sample is associated with a label $+1/-1$ indicating whether gene expression is high or low respectively. The gene expression values were discretized using the median of gene expressions across all genes for a particular cell-type.

3.4.2 Baselines

We compare DeepChrome to two baseline studies, [69] which uses Support Vector Machines (SVM) and [71] which uses a Random Forest Classifier. Their implementation strategies are as follows:

- *SVM ([69]):* The authors selected 160 bins from regions flanking the gene TSS and TTS. Each bin position uses a separate SVM classification model, resulting in 160 different models in total. This gave insights into important bin positions for classifying gene expression as high or low. Following this bin-specific model strategy, we provide results for performance of the best bin (SVM Best Bin) along with average performance across all bins (SVM Avg) in Section 5.1 and in Figure 3.3.

- *Random Forest Classifier ([71]):* In this study, bins were selected from regions flanking the TSS, TTS, and gene body. This study selected the bin values having the highest correlation with gene expression as "best bins". A matrix with all genes and best bins for each histone modification signal was used as input into the model to predict gene labels $(+1/-1)$ as output. Since this baseline performs feature selection using the best bin strategy, our experiment uses the best-bin Random Forest performance as a baseline in Figure 3.3.

We implemented these baselines using the python based scikit-learn ([81]) package.

3.4.3 Hyperparameter tuning

For each cell type, our sample set of total 19802 genes was divided into 3 separate, but equal size folds: training (6601 genes), validation (6601 genes) and test (6600 genes) sets. We trained DeepChrome using the following hyperparameters: filter size ($k = \{10, 5\}$), number of convolution filters ($N_{out} = \{20, 50, 100\}$) and pool size for maxpooling ($m = \{2, 5\}$). Table 3.3 presents the validation set results for tuning different combinations of kernel size k and pool size m. k denotes the local neighborhood representations of flanking

bins. m represents selected whole regions in our CNN model. We report the maximum, minimum and mean AUC scores obtained across 56 cell types. Performances of models using these different hyperparameter values did not vary significantly (p-value$\sim$ 0.92) from each other. We also trained a deeper model with 2 convolution layers and observed no significant (p-value= 0.939) increase in performance.

- We selected $k = 10$, $N_{out} = 50$, and $m = 5$ for training the final CNN models based on highest Max. and Min. AUC scores in Table 3.3. The number of hidden units chosen for the two multilayer perceptron layers were 625 and 125, respectively. We trained the model for 100 epochs and observed that it converged early (around 15-20 epochs).

- For the SVM implementation, an RBF kernel was selected and the model was trained on varying hyperparameter values of $C \in \{0.01, 0.1, 1, 10, 100, 1000\}$ and $\gamma \in \{0.01, 0.1, 1, 2\}$. The C parameter balances the trade-off between misclassification of training examples and simplicity of the decision surface, while the γ parameter defines the extent of influence of a single training sample.

- For the Random Forest Classifier implementation, we varied the number of decision trees, $n_d \in \{10, 20, ..., 200\}$ trained in each model.

All the above models were trained on the training set, and the parameters for testing were selected based on their results on the validation set. We then applied the selected models on the test dataset. The AUC scores [6] (performance metric) are reported in Section 5.1.

3.5 Results

3.5.1 Performance Evaluation

The bar graph in Figure 3.3 compares the performance of DeepChrome and three baselines on test data set for gene expression classification across 56 different cell-types (or tasks). DeepChrome (Average AUC = 0.80) outperforms the baselines for all the cell types shown along the X-axis. As discussed earlier, Cheng et al. [69] implement a different SVM model for each bin position. Therefore, we report both average AUC score for all the bins (SVM Avg) as well as the best AUC score among all bins (SVM Best Bin). 'SVM Best Bin'

[6] Area Under Curve (AUC) score from Receiver Operating Characteristic (ROC) curve is interpreted as the probability that a randomly selected "event" will be regarded with greater suspicion (in terms of its continuous measurement) than a randomly selected "non-event". AUC score ranges between 0 and 1, where values closer to 1 indicate more successful predictions.

Table 3.3: Results on validation set (6601 genes) during tuning across different combinations of kernel size k and pool size m.

k captures the local neighborhood representations of bins and m combines the important representations across whole regions for our CNN model. We report the maximum, minimum and mean AUC score obtained across 56 cell types (or tasks). The best performing values of $k = 10$ and $m = 5$ (highest Max. and Min. AUC scores) were selected evaluating test performance of DeepChrome.

Kernel Size, Pool Size (k, m)	AUC Scores (Validation Set)		
	Max	Min	Mean
(5,2)	0.94	0.65	0.77
(5,5)	0.94	0.65	0.77
(10,2)	0.94	0.65	0.76
(10,5)	0.94	0.66	0.77

(Average AUC = 0.75) gives better results than 'SVM Avg' (Average AUC = 0.66). However, its AUC scores are still lower than those of DeepChrome. Random Forest gives the worst performance (Average AUC =0.59). Additionally, we observe that the performances of all three models vary across different cell types and follow a similar trend. For some cell types, like E123, the prediction task resulted in higher AUC scores among all models than other cell types. Two hypotheses can explain the disparity among the performances across cell types:

Data-driven: The datasets were downloaded for 56 different cell types archived by the REMC database [1]. Here, the HM ChIP-Seq signals were consolidated across different experiments for the same cell type. Therefore, while E123 (K562) is a well-studied cell type with multiple experiments profiling the HMs, E112 (Thymus) has 1-2 ChIP-Seq experiments per HM. On consolidation, E123 will have an advantage over E112 with regards to data quality as combining more experiments will get rid of noisy signals in the data.

Biology-driven: These differences could also arise from the heterogeneity in the cells. Cell-level heterogeneity is observed as phenotypic differences arising within genetically uniform cell populations, due to different HM signals, etc. For example, E112 (Thymus) belongs to the immune system, which has known heterogeneity within the cell populations [82]. Whereas, E123 (K562) is a homogeneous cell line, which would result in consistent HM signals.

3.5.2 Validating the influence of bin positions on prediction

Cheng et al. [69] obtained predictions for each bin (due to bin-specific strategy) and reported that, on average, the best AUC scores were obtained from bins that are close to the TSS. Figure 3.4 (a) shows that

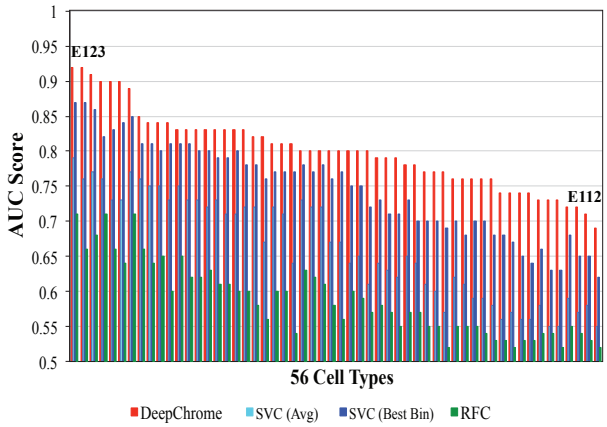

Figure 3.3: Performance Evaluation on Test Set

The bar graph represents AUC scores of DeepChrome versus state-of-the-art baseline models for 56 cell types (i.e. 56 different classification tasks). The results have been arranged from best performing cell type (E123) to the worst performing cell type (E112) for the test set (6600 genes). DeepChrome (Average AUC = 0.80) consistently outperforms both SVM (Average AUC: SVM Best Bin = 0.75 and SVM Avg = 0.66) and Random Forest Classifier (Average AUC = 0.59) for the task of binary classification of gene expression. SVM based baseline has a separate model for each bin (bin specific model), thus, results for both average AUC scores across all bins (SVM Avg) and best performing AUC score among the bins (SVM Best Bin) are presented.

our implementation of this SVM baseline confirms this observation. Since our convolutional network makes a prediction on the entire flanking region (i.e. all the bins at once), we cannot evaluate the AUC for each individual bin. However, we can roughly determine which bins are the most influential for a specific gene prediction. To do this, we look at the strongest activations among the output of the convolution step (the feature map, as shown in Figure 3.2). Since the column in the feature map corresponds to the bins in the input region, we can simply look at the feature map values to determine which bin positions are most influential for that prediction. To validate our model, we ran all of our test samples through a trained deep network, and took the average of all the feature maps across all 56 models. Figure 3.4(b) shows that bins near the center, closer to TSS, are assigned with higher values by the convolution layers. This indicates that DeepChrome maintains similar trends as observed by Cheng et al. [69]. This trend indicates histone modification signals of bins that are closer to TSS are more influential in gene predictions.

3.5.3 Visualizing Combinatorial Interactions among Histone Modifications

In order to interpret the combinatorial interactions among histone modifications, we present a visualization technique in Section 3.3. Figure 3.5 presents four visualization results from DeepChrome on four cell types with high AUC scores. Each visualization result is a heat-map which shows the histone modification combinatorial

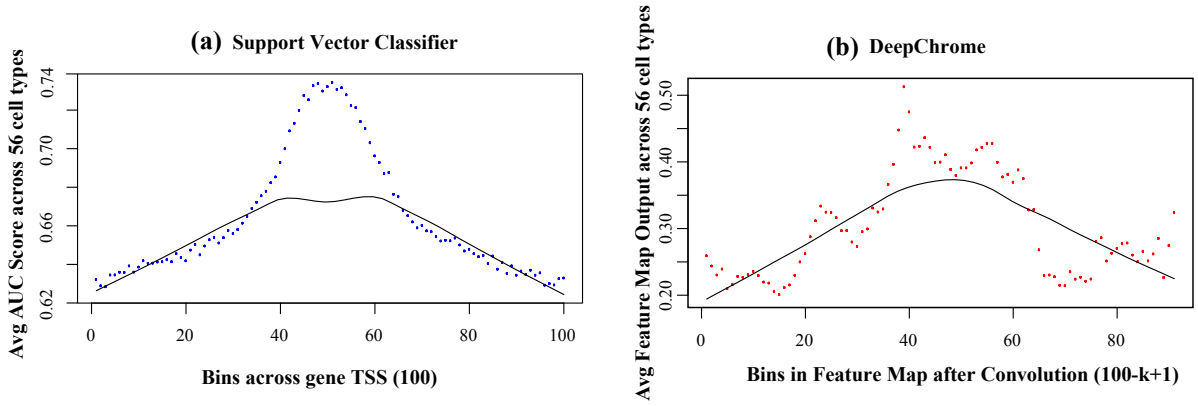

Figure 3.4: Validating the influence of positions for gene expression classification.

Cheng et al. [69] reported that the bin positions closer to the transcription start site (TSS) of each gene are more important when predicting gene expression. This is confirmed by our implementation of this bin-specific baseline model in (a). For each bin position, it shows the mean AUC score across all the cell types. In (b) we plot the filter outputs from the convolution layer of DeepChrome model. For each bin, its value has been averaged across all filters and cell-types. The solid lines represent the best-curve fit to the data points plotted in the figures. The trends for both (a) and (b) are similar.

pattern that is best representative of high (label = $+1$) or low (label = -1) gene expression. Note that this is different than Section 5.2 where we validated the importance of bin positions in general, rather than the combinatorial interactions for a specific class. The values in the heatmaps are within the range $[0, 1]$, representing how important a particular bin is for prediction of the class of interest. A threshold of 0.25 was selected to filter "active", or important, bins that are most influential for a particular classification. We calculated the frequency count of active bins for each histone modification. Histone marks with high frequency counts ($>$ mean frequency count across all histone marks) are considered to be strongly affecting the gene expression to become high or low. As expected, we observe a relationship among promoter and structural histone modification marks (H_{prom} and H_{struct}) for 47 out of 56 (84%) cell-types when gene expression is high. Similarly, we observe an opposite trend with repressor marks (H_{reprA} and H_{reprA}) showing combinatorial relationship for 50 out of 56 (89%) cell-types, when gene expression is low. In other words, our model automatically learns that in order to classify a high or low gene expression, there needs to be high counts among promoter marks, or repressor marks, respectively.

Next, we validated our visualization results with the findings in previous studies. Both of our baseline papers, [69] and [71], showed that there is a combinatorial correlation between H3K4me3 (H_{prom}) and H3K36me3 (H_{struct}). This pattern can be seen in Figure 3.5 for high gene expression cases. Similarly, Dong et al. [71] also reported a combinatorial correlation between H_{prom} (H3K4me3) and distal promoter mark (H_{enhc}), which is also validated by the DeepChrome visualization for 35 out of 56 cell-types (62.5%). In addition,

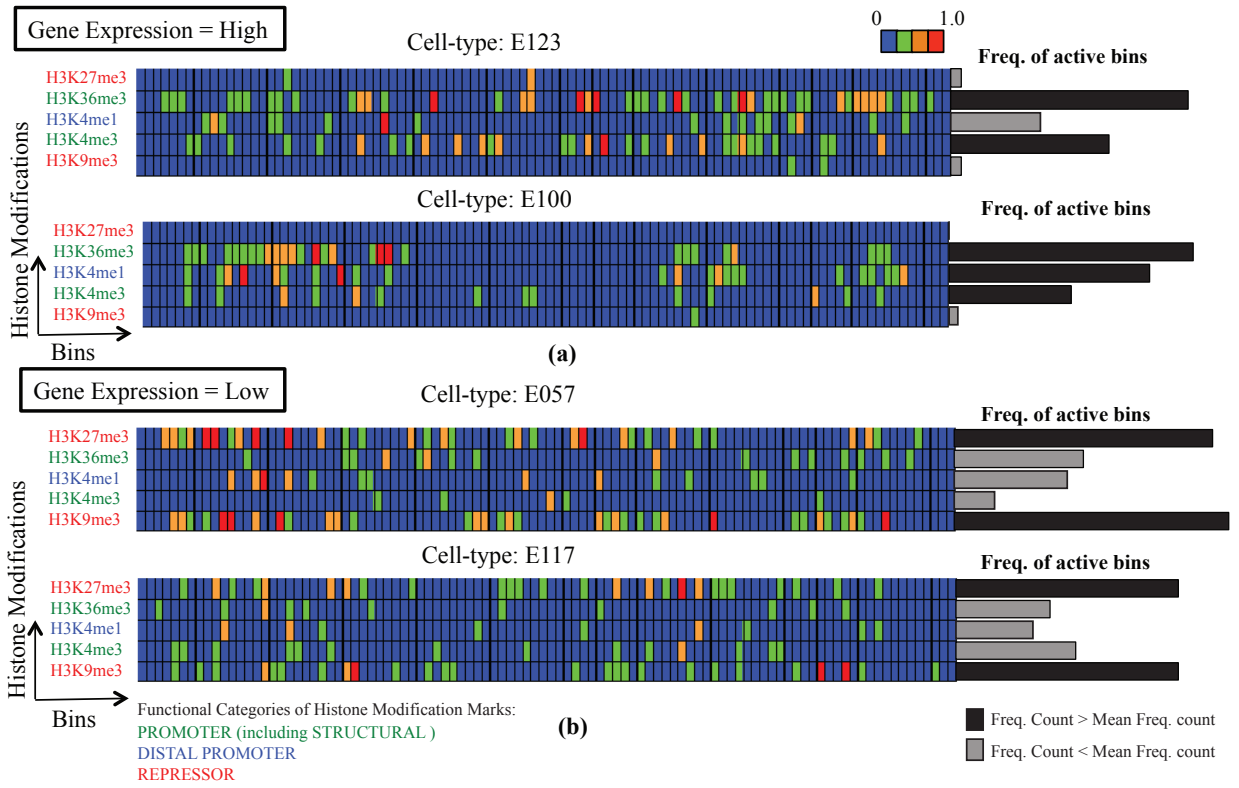

Figure 3.5: DeepChrome visualization

Four examples of feature maps generated by our optimization technique from four trained models. The scores in these feature maps are $\in [0, 1]$ and a threshold of 0.25 was selected to indicate "active" (or important) bins. The bar graph represents the count of active bins for each histone modification. Higher frequency count ($>$ mean frequency count across all histone marks) indicates greater influence of the histone modification mark in prediction of gene expression labels. Multiple marks with high frequency count are considered to be combinatorially affecting the gene expression to become high or low. (a) As expected, we observe a relationship among promoter and structural histone modification marks (H_{prom} and H_{struct}) when gene expression is high. (b) Similarly, we observe an opposite trend with repressor marks (H3K9me3 and H3K27me3) showing combinatorial relationship, when gene expression is low. These pattern maps not only support previous quantitative observations in [69] and [71], but also provide novel insights that are supported by recent biological studies. For example, a recent study by Boros et al. [83] has reported evidence of coexistence of H_{reprA} and H_{reprB} modifications in gene silencing.

experimental studies have shown that these promoter marks play a role in the activation of genes, and this trend is seen in our visualization when the assigned label is $+1$.

Another combinatorial pattern that we noticed in the majority of cell-types (89%, i.e 50 out of 56 cell-types) was that of H_{reprA} (polycomb repressor) and H_{reprB} (heterochromatin repressor) for low gene expression case (label$=-1$). We found this observation in multiple recent biological studies such as [83]. This study reported that these two repressor marks coexist and cooperate in gene silencing. With almost no expert knowledge, we were able to find and visualize this combination through DeepChrome. To our knowledge, none of the previous computational studies have reported this combinatorial effect between H3K27me3 (H_{reprA}) and

H3K9me3 (H_{reprB}). In short, the DeepChrome visualization technique provides the potential to learn novel insights into combinatorial relationships among histone modifications for gene regulation.

In summary, DeepChrome opens multiple new avenues for studying and exploration of genetic regulation via epigenetic factors. This is made possible due to deep learning's ability to handle a large amount of existing data as well as to automatically extract important features and complex interactions, providing us with important insights. Techniques like DeepChrome hold the potential to bring us one step closer to properly investigating gene regulation mechanisms, which in turn can lead to understanding of genetic diseases.

Chapter 4

Towards Interpretability of Deep Neural Networks

4.1 Why Interpretability is Important?

Owing to their state-of-the-art performance, DNNs are being integrated into various intelligent systems as critical components, e.g., speech recognition devices, etc. Their high performance is attributed to the stacked and dense layers that encode feature representations as numerical weights of the various node to node connections. These dense networks have complex formulation, which leads to unclear internal working mechanisms. Hence DNNs are viewed as "black-boxes' in the literature. Without a clear understanding of the inner workings of the network, the training of DNNs is mostly empirically driven. Furthermore, their application to real-world systems is also limited as we are unable to explain the reasons behind the decisions or actions to human users. While this limitation might not be a significant issue for image classification systems, in the biomedical domain, it is a major roadblock for integration of DNNs with diagnostic systems. For example, in case of medical image classification, a radiologist would require an explicit interpretable output from a DNN so that he or she can integrate its decision with his or her diagnosis. Also, interpretable DNNs can allow us to explain the reason behind a wrong prediction that might have a high impact on the outcome of the model. Therefore, development of interpretable DNN models is crucial so that users can understand, trust and interact efficiently with intelligent systems.

Table 4.1: Comparison of previous studies with AttentiveChrome.

AttentiveChrome is the only model that exhibits all 8 desirable properties.

Computational Study	Unified	Non-linear	Bin-Info	Representation Learning		Prediction	Feature Inter.	Interpretable
				Neighbor Bins	Whole Region			
Linear Regression ([66])	×	×	×	×	✓	✓	×	✓
Support Vector Machine ([69])	×	✓	Bin-specific	×	✓	✓	✓	×
Random Forest ([71])	×	✓	Best-bin	×	✓	✓	×	×
Rule Learning ([73])	×	✓	×	×	✓	×	✓	✓
DeepChrome-CNN [6]	✓	✓	Automatic	✓	✓	✓	✓	×
AttentiveChrome	✓	✓	Automatic	✓	✓	✓	✓	✓

In this chapter we propose an attention-based deep learning model, AttentiveChrome, that learns to predict the expression of a gene from an input of histone modification signals covering the gene's neighboring DNA region. By using a hierarchy of multiple LSTM modules, AttentiveChrome can discover interactions among signals of each chromatin mark, and simultaneously learn complex dependencies among different marks. Two levels of "soft" attention mechanisms are trained, (1) to attend to the most relevant regions of a chromatin mark, and (2) to recognize and attend to the important marks. Through predicting and attending in one unified architecture, AttentiveChrome allows users to understand how chromatin marks control gene regulation in a cell. In summary, this work makes the following contributions:

- AttentiveChrome provides more accurate predictions than state-of-the-art baselines. Using datasets from REMC, we evaluate AttentiveChrome on 56 different cell types (tasks).

- We validate and compare interpretation scores using correlation to a new mark signal from REMC (not used in modeling). AttentiveChrome's attention scores provide a better interpretation than state-of-the-art methods for visualizing deep learning models.

- AttentiveChrome can model highly modular inputs where each module is highly structured. AttentiveChrome can explain its decisions by providing "what" and "where" the model has focused on. This flexibility and interpretability make this model an ideal approach for many real-world applications.

In the following sections, we denote vectors with bold font and matrices using capital letters. To simplify notation, we use "HM" as a short form for the term "histone modification".

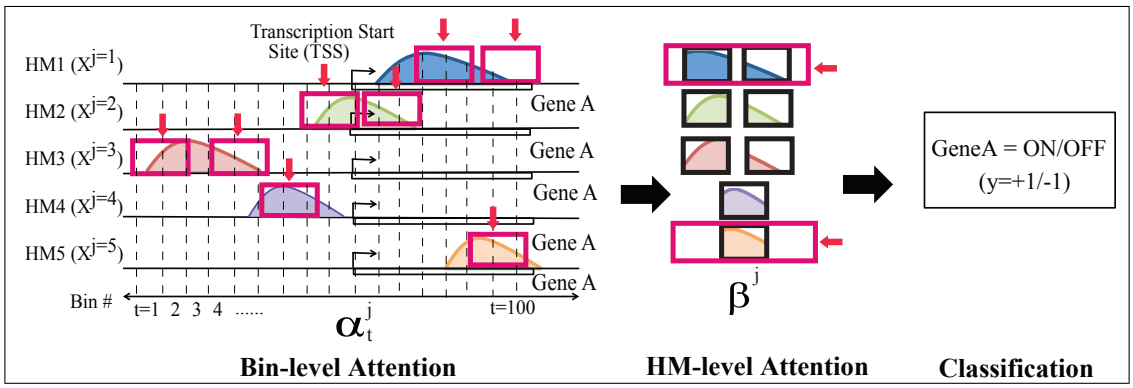

Figure 4.1: Overview of the proposed AttentiveChrome framework

It includes 5 important parts: (1) Bin-level LSTM encoder for each HM mark; (2) Bin-level α-Attention across all bin positions of each HM mark; (3) HM-level LSTM encoder encoding all HM marks; (4) HM-level β-Attention among all HM marks; (5) the final classification.

4.2 Approach

4.2.1 Input and Output Formulation for the Task

We use the same feature inputs and outputs as done previously in DeepChrome ([6]). Following Cheng et al. [69], the gene expression prediction is formulated as a binary classification task whose output represents if the gene expression of a gene is high(+1) or low(-1). As shown in Figure 4.1, the input feature of a sample (a particular gene) is denoted as a matrix $\mathbf{X}$ of size $M \times T$. Here M denotes the number of HM marks we consider in the input. T is the total number of bin positions we take into account from the neighboring region of a gene's TSS site on the genome. We refer to this region as the 'gene region' in the rest of the chapter. $\mathbf{x}^j$ denotes the j-th row vector of $\mathbf{X}$ whose elements are sequentially structured (signals from the j-th HM mark) $j \in \{1, ..., M\}$. x_t^j in matrix $\mathbf{X}$ represents the signal from the t-th bin of the j-th HM mark. $t \in \{1, ..., T\}$. We assume our training set D contains N_{tr} labeled pairs. We denote the n-th pair as $(\mathbf{X}^{(n)}, y^{(n)})$, $\mathbf{X}^{(n)}$ is a matrix of size $M \times T$ and $y^{(n)} \in \{-1, +1\}$, where $n \in \{1, ..., N_{tr}\}$.

4.2.2 Long Short-Term Memory (LSTM) Networks

Recurrent neural networks (RNNs) have been designed for modeling sequential data samples and are used widely in sequential data application tasks such as natural language processing. RNNs are advantageous over

CNNs because they can capture the complete set of dependencies among spatial positions in a sequential sample.

Given an input matrix $\mathbf{X}$ of size $n_{in} \times T$, an RNN produces a matrix $\mathbf{H}$ of size $d \times T$, where n_{in} is the input feature size, T is the input feature length, and d is the RNN embedding size. At each timestep $t \in [1..T]$, an RNN takes an input column vector $\mathbf{x_t} \in \mathbb{R}^{n_{in}}$ and the previous hidden state vector $\mathbf{h_{t-1}} \in \mathbb{R}^d$ and produces the next hidden state $\mathbf{h_t}$ by applying the following recursive operation:

$$\mathbf{h}_t = \sigma(\mathbf{W}x_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}), \tag{4.1}$$

where $\mathbf{W}, \mathbf{U}, \mathbf{b}$ are the trainable parameters of the model, and σ is an element-wise nonlinearity function. Due to their recursive nature, RNNs can model the full conditional distribution of any sequential data and find dependencies over time. To handle "vanishing gradient" issue of training basic RNNs, Hochreiter et al. [84] proposed an RNN variant called the Long Short-term Memory (LSTM) network,which can handle long term dependencies by using gating functions. These gates can control when information is written to,

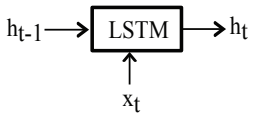

Figure 4.2: A simple representation of an LSTM module.

read from, and forgotten. Specifically, LSTM "cells" take inputs $\mathbf{x}_t, \mathbf{h}_{t-1}$, and $\mathbf{c}_{t-1}$, and produce $\mathbf{h}_t$, and $\mathbf{c}_t$:

$$\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}^i\mathbf{x}_t + \mathbf{U}^i\mathbf{h}_{t-1} + \mathbf{b}^i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}^f\mathbf{x}_t + \mathbf{U}^f\mathbf{h}_{t-1} + \mathbf{b}^f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}^o\mathbf{x}_t + \mathbf{U}^o\mathbf{h}_{t-1} + \mathbf{b}^o) \\
\mathbf{g}_t &= tanh(\mathbf{W}^g\mathbf{x}_t + \mathbf{U}^g\mathbf{h}_{t-1} + \mathbf{b}^g) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot tanh(\mathbf{c}_t)
\end{aligned}$$

where $\sigma(\cdot)$, $tanh(\cdot)$, and $\odot$ are element-wise sigmoid, hyperbolic tangent, and multiplication functions, respectively. $\mathbf{i}_t$, $\mathbf{f}_t$, and $\mathbf{o}_t$ are the input, forget, and output gates, respectively.

4.2.3 An End-to-End Deep Architecture for Predicting and Attending Jointly

AttentiveChrome learns to predict the expression of a gene from an input of HM signals covering its gene region. First, the signals of each HM mark are fed into a separate LSTM network to encode the spatial dependencies among its bin signals, and then another LSTM is used to model how multiple factors work together for predicting gene expression. Two levels of "soft" attention mechanisms are trained and dynamically predicted for each gene: (1) to attend to the most relevant positions of an HM mark, and (2) then to recognize and attend to the relevant marks. In summary, AttentiveChrome consists of five main modules (Figure 4.4): (1) Bin-level LSTM encoder for each HM mark; (2) Bin-level Attention on each HM mark; (3) HM-level LSTM encoder encoding all HM marks; (4) HM-level Attention over all the HM marks; (5) the final classification module. We describe the details of each component as follows:

4.2.4 Bin-Level Encoder Using LSTMs

For a gene of interest, the j-th row vector, $\mathbf{x}^j$, from $\mathbf{X}$ includes a total of T elements that are sequentially ordered along the genome coordinate. Considering the sequential nature of such signal reads, we treat each element (essentially a bin position) as a 'time step' and use a bidirectional LSTM to model the complete dependencies among elements in $\mathbf{x}^j$. A bidirectional LSTM contains two LSTMs, one in each direction (see Figure 4.4(c)). It includes a forward $\overrightarrow{LSTM}^j$ that models $\mathbf{x}^j$ from x_1^j to x_T^j and a backward $\overleftarrow{LSTM}^j$ that models from x_T^j to x_1^j. For each position t, the two LSTMs output $\overrightarrow{\mathbf{h}_t^j}$ and $\overleftarrow{\mathbf{h}_t^j}$, each of size d. $\overrightarrow{\mathbf{h}_t^j} = \overrightarrow{LSTM}^j(x_t^j)$ and $\overleftarrow{\mathbf{h}_t^j} = \overleftarrow{LSTM}^j(x_t^j)$. The final embedding vector at the t-th position is the concatenation $\mathbf{h}_t^j = [\overrightarrow{\mathbf{h}_t^j}, \overleftarrow{\mathbf{h}_t^j}]$. By coupling these LSTM-based HM encoders with the final classification, they can learn to embed each HM mark by extracting the dependencies among bins that are essential for the prediction task.

4.2.5 Bin-Level Attention, α-attention

Although the LSTM can encode dependencies among the bins, it is difficult to determine which bins are *most important* for prediction from the LSTM. To automatically and adaptively highlight the most relevant bins for each sample, we use "soft" attention to learn the importance weights of bins. This means when representing j-th HM mark, AttentiveChrome follows a basic concept that not all bins contribute equally to the encoding of the entire j-th HM mark. The attention mechanism can help locate and recognize those

bins that are important for the current gene sample of interest from j-th HM mark and can aggregate those important bins to form an embedding vector. This extraction is implemented through learning a weight vector $\boldsymbol{\alpha}^j$ of size T for the j-th HM mark. For $t \in \{1, ..., T\}$, α_t^j represents the importance of the t-th bin in the j-th HM. It is computed as: $\boldsymbol{\alpha}_t^j = \frac{exp(\mathbf{W}_b \mathbf{h}_t^j)}{\sum_{i=1}^{T} exp(\mathbf{W}_b \mathbf{h}_i^j)}$.

α_t^j is a scalar and is computed by considering all bins' embedding vectors $\{\mathbf{h}_1^j, \cdots, \mathbf{h}_T^j\}$. The context parameter $\mathbf{W}_b$ is randomly initialized and jointly learned with the other model parameters during training. Our intuition is that through $\mathbf{W}_b$ the model will automatically learn the context of the task (e.g., type of a cell) as well as the positional relevance to the context simultaneously. Once we have the importance weight of each bin position, we can represent the entire j-th HM mark as a weighted sum of all its bin embeddings: $\mathbf{m}^j = \sum_{t=1}^{T} \alpha_t^j \times \mathbf{h}_t^j$. Essentially the attention weights α_t^j tell us the relative importance of the t-th bin in the representation $\mathbf{m}^j$ for the current input $\mathbf{X}$ (both $\mathbf{h}_t^j$ and α_t^j depend on $\mathbf{X}$).

4.2.6 HM-Level Encoder Using Another LSTM

We aim to capture the dependencies among HMs as some HMs are known to work together to repress or activate gene expression [83]. Therefore, next we model the joint dependencies among multiple HM marks (essentially, learn to represent a set). Even though there exists no clear order among HMs, we assume an imagined sequence as $\{HM_1, HM_2, HM_3, ..., HM_M\}$ [1]. We implement another bi-directional LSTM encoder, this time on the imagined sequence of HMs using the representations $\mathbf{m}^j$ of the j-th HMs as LSTM inputs (Figure 4.4(e)). Setting the embedding size as d', this set-based encoder, we denote as $LSTM_s$, encodes the j-th HM as: $\mathbf{s}^j = [\overrightarrow{LSTM_s}(\mathbf{m}^j), \overleftarrow{LSTM_s}(\mathbf{m}^j)]$. Differently from $\mathbf{m}^j$, $\mathbf{s}^j$ encodes the dependencies between the j-th HM and other HM marks.

4.2.7 HM-Level Attention, β-attention

Now we want to focus on the important HM markers for classifying a gene's expression as high or low. We do this by learning a second level of attention among HMs. Similar to learning α_t^j, we learn another set of weights β^j for $j \in \{1, \cdots, M\}$ representing the importance of HM^j. β^i is calculated as: $\beta^j = \frac{exp(\mathbf{W}_s \mathbf{s}^j)}{\sum_{i=1}^{M} exp(\mathbf{W}_s \mathbf{s}^i)}$. The *HM-level* context parameter $\mathbf{W}_s$ learns the context of the task and learns how HMs are relevant to that

[1] We tried several different architectures to model the dependencies among HMs, and found no clear ordering.

context. $\mathbf{W}_s$ is randomly initialized and jointly trained. We encode the entire "gene region" into a hidden representation $\mathbf{v}$ as a weighted sum of embeddings from all HM marks: $\mathbf{v} = \sum_{j=1}^{M} \beta^j \mathbf{s}^j$. We can interpret the learned attention weight β^i as the relative importance of HM^i when blending all HM marks to represent the entire gene region for the current gene sample $\mathbf{X}$.

4.2.8 Training AttentiveChrome End-to-End

The vector $\mathbf{v}$ summarizes the information of all HMs for a gene sample. We feed it to a simple classification module f (Figure 4.4(f)) that computes the probability of the current gene being expressed high or low: $f(\mathbf{v}) = \text{softmax}(\mathbf{W}_c\mathbf{v} + b_c)$. $\mathbf{W}_c$ and b_c are learnable parameters. Since the entire model, including the attention mechanisms, is differentiable, learning end-to-end is trivial by using backpropagation [76]. All parameters are learned together to minimize a negative log-likelihood loss function that captures the difference between true labels y and predicted scores from $f(.)$.

AttentiveChrome Forward Propagation algorithm is presented in Algorithm box 1, while Figure 3.2 presents the overview of the proposed AttentiveChrome in detail.

Algorithm 1 AttentiveChrome: Forward Propagation

Require: X ▷ Size: $M \times T$

procedure CLASSIFICATION(X)
  $\{x_1^t, x_2^t, \dots x_j^t\} \leftarrow X$ ▷ Size: $1 \times T$, $t \in \{1, \dots T\}$ and $j \in \{1, \dots M\}$
  $\mathbf{m}^j \leftarrow BinLevelAttention(x_t^j)$
  $\mathbf{v} \leftarrow HMLevelAttention(\mathbf{m}^j)$
  $y \leftarrow MultiLayerPerceptron(\mathbf{v})$
return y
procedure BIN-LEVEL ATTENTION(x_t^j)
  for $j \in \{1, \dots M\}$ **do** ▷ Run in Parallel
    $\overrightarrow{\mathbf{h}_t^j} \leftarrow \overrightarrow{LSTM}^j(x_t^j)$ ▷ Bi-directional LSTM
    $\overleftarrow{\mathbf{h}_t^j} \leftarrow \overleftarrow{LSTM}^j(x_t^j)$
    $\mathbf{h}_t^j \leftarrow [\overrightarrow{\mathbf{h}_t^j}, \overleftarrow{\mathbf{h}_t^j}]$.
    $\boldsymbol{\alpha}_t^j \leftarrow \frac{exp(\mathbf{W}_b\mathbf{h}_t^j)}{\sum_{i=1}^{T} exp(\mathbf{W}_b\mathbf{h}_i^j)}$ ▷ Size: $1 \times T$ for each $j \in \{1, \dots M\}$
    $\mathbf{m}^j \leftarrow \sum_{t=1}^{T} \alpha_t^j \times \mathbf{h}_t^j$
  return $\mathbf{m}^j$
procedure HM-LEVEL ATTENTION($\mathbf{m}^j$)
  $\mathbf{s}^j \leftarrow [\overrightarrow{LSTM_s}(\mathbf{m}^j), \overleftarrow{LSTM_s}(\mathbf{m}^j)]$
  $\beta^j \leftarrow \frac{exp(\mathbf{W}_s\mathbf{s}^j)}{\sum_{i=1}^{M} exp(\mathbf{W}_s\mathbf{s}^i)}$ ▷ Size: $1 \times M$
  $\mathbf{v} \leftarrow \sum_{j=1}^{M} \beta^j \mathbf{s}^j$
  return $\mathbf{v}$

4.3 Experimental Setup

4.3.1 Dataset

We use the same dataset as in DeepChrome with gene expression levels and signal data of five core HM marks for 56 different cell types archived by the REMC database [1]. Each dataset contains information about both the location and the signal intensity for a mark measured across the whole genome. The selected five core HM marks have been uniformly profiled across all 56 cell types in the REMC study [1]. We revisit the naming of these five HM marks include: H3K27me3 as H_{reprA}, H3K36me3 as H_{struct}, H3K4me1 as H_{enhc}, H3K4me3 as H_{prom}, and H3K9me3 as H_{reprB}. HMs H_{reprA} and H_{reprB} are known to repress the gene expression, H_{prom} activates gene expression, H_{struct} is found over the gene body, and H_{enhc} sometimes helps in activating gene expression.

4.3.2 Model Variations and Two Baselines

In Section 4.2, we introduced three main components of AttentiveChrome to handle the task of predicting gene expression from HM marks: LSTMs, attention mechanisms, and hierarchical attention. To investigate the performance of these components, our experiments compare multiple AttentiveChrome model variations plus two standard baselines.

- DeepChrome [6]: The temporal (1-D) CNN model used by Singh et al. [6] for the same classification task. This study did not consider the modular property of HM marks.
- LSTM: We directly apply an LSTM on the input matrix $\mathbf{X}$ without adding any attention. This setup does not consider the modular property of each HM mark, that is, we treat the signals of all HMs at t-th bin position as the t-th input to LSTM.
- LSTM-Attn: We add one attention layer on the baseline LSTM model over input $\mathbf{X}$. This setup does not consider the modular property of HM marks.
- CNN-Attn: We apply one attention layer over the CNN model from DeepChrome [6], after removing the max-pooling layer to allow bin-level attention for each bin. This setup does not consider the modular property of HM marks.

- LSTM-α, β: As introduced in Section 4.2, this model uses one LSTM per HM mark and add one α-attention per mark. Then it uses another level of LSTM and β-attention to combine HMs.
- CNN-α, β: This considers the modular property among HM marks. We apply one CNN per HM mark and add one α-attention per mark. Then it uses another level of CNN and β-attention to combine HMs.
- LSTM-α: This considers the modular property of HM marks. We apply one LSTM per HM mark and add one α-attention per mark. Then, the embedding of HM marks is concatenated as one long vector and then fed to a 2-layer fully connected MLP.

We use datasets across 56 cell types, comparing the above methods over each of the 56 different tasks.

Table 4.2: AUC score performances for different variations of AttentiveChrome and baselines

	Baselines		AttentiveChrome Variations				
Model	DeepChrome (CNN) [6]	LSTM	CNN-Attn	CNN-α, β	LSTM-Attn	LSTM-α	LSTM-α, β
Mean	0.8008	0.8052	0.7622	0.7936	0.8100	**0.8133**	0.8115
Median	0.8009	0.8036	0.7617	0.7914	0.8118	**0.8143**	0.8123
Max	**0.9225**	0.9185	0.8707	0.9059	0.9155	0.9218	0.9177
Min	0.6854	0.7073	0.6469	0.7001	**0.7237**	0.7250	0.7215
Improvement over DeepChrome [6] (out of 56 cell types)		36	0	16	49	**50**	49

4.3.3 Model Hyperparameters

For AttentiveChrome variations, we set the bin-level LSTM embedding size d to 32 and the HM-level LSTM embedding size as 16. Since we implement a bi-directional LSTM, this results in each embedding vector $\mathbf{h}_t$ as size 64 and embedding vector $\mathbf{m}_j$ as size 32. Therefore, we set the context vectors, $\mathbf{W_b}$ and $\mathbf{W_s}$, to size 64 and 32 respectively.[2]

[2]We can view $\mathbf{W_b}$ as 1×64 matrix.

4.4 Results

4.4.1 Performance Evaluation

Table 4.2 compares different variations of AttentiveChrome using summarized AUC scores across all 56 cell types on the test set. We find that overall the LSTM-attention based models perform better than CNN-based and LSTM baselines. CNN-attention model gives worst performance. To add the bin-level attention layer to the CNN model, we removed the max-pooling layer. We hypothesize that the absence of max-pooling is the cause behind its low performance. LSTM-α has better empirical performance than the LSTM-α, β model. We recommend the use of the proposed AttentiveChrome LSTM-α, β (from here on referred to as AttentiveChrome) for hypothesis generation because it provides a good trade-off between AUC and interpretability. Also, while the performance improvement over DeepChrome [6] is not large, AttentiveChrome is better as it allows interpretability to the "black box" neural networks.

Table 4.3: Pearson Correlation values between weights assigned for H_{prom} (active HM) by different visualization techniques and H_{active} read coverage (indicating actual activity near "ON" genes) for predicted "ON" genes across three major cell types.

Viz. Methods	H1-hESC	GM12878	K562
α Map (LSTM-α)	0.8523	**0.8827**	**0.9147**
α Map (LSTM-α, β)	**0.8995**	0.8456	0.9027
Class-based Optimization (CNN)	0.0562	0.1741	0.1116
Saliency Map (CNN)	0.1822	-0.1421	0.2238

4.4.2 Using Attention Scores for Interpretation

Unlike images and text, the results for biology are hard to interpret by just looking at them. Therefore, we use additional evidence from REMC as well as introducing a new strategy to qualitatively and quantitatively evaluate the bin-level attention weights or α-map LSTM-α model and AttentiveChrome. To specifically validate that the model is focusing its attention at the right bins, we use the read counts of a new HM signal - H3K27ac from REMC database. We represent this HM as H_{active} because this HM marks the region that is active when the gene is "ON". H3K27ac is an important indication of activity in the DNA regions and is a good source to validate the results. We did not include H3K27ac Mark as input because it has not been profiled for all 56 cell types we used for prediction. However, the genome-wide reads of this HM mark are available for three important cell types in the blood lineage: H1-hESC (stem cell), GM12878 (blood cell), and K562 (leukemia cell). We, therefore, chose to compare and validate interpretation in these three cell

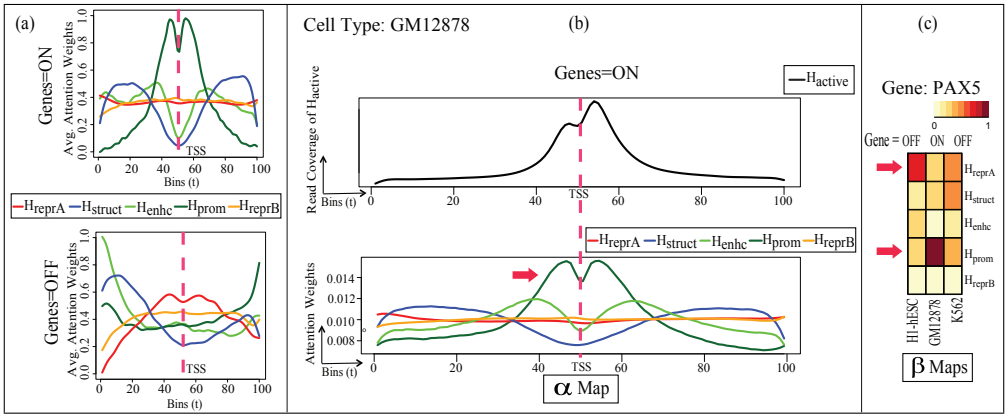

Figure 4.3: Visualization results of Attentivechrome

(a) Bin-level attention weights (α_t^j) from AttentiveChrome averaged for all genes when predicting gene=ON and gene=OFF in GM12878 cell type. (b) Top: Cumulative H_{active} signal across all active genes. Bottom: Plot of the bin-level attention weights (α_t^j). These weights are averaged for gene=ON predictions. H_{prom} weights are concentrated near the TSS and corresponds well with the H_{active} indicating actual activity near the gene. This indicates that AttentiveChrome is focusing on the correct bin positions for this case (c) Heatmaps visualizing the HM-level weights (β^j), with $j \in \{1, ..., 5\}$ for an important differentially regulated gene (PAX5) across three blood lineage cell types: H1-hESC (stem cell), GM12878 (blood cell), and K562 (leukemia cell). The trend of HM-level β^j weights for PAX5 have been verified through biological literature.

types. This HM signal has not been used at any stage of the model training or testing. We use it solely to analyze the visualization results.

We use the average read counts of H_{active} across all 100 bins and for all the active genes (gene=ON) in the three selected cell types to compare different visualization methods. We compare the attention α-maps of the best performing LSTM-α and AttentiveChrome models with the other two popular visualization techniques: (1) the Class-based optimization method and (2) the Saliency map applied on the baseline DeepChrome-CNN model. We take the importance weights calculated by all visualization methods for our active input mark, H_{prom}, across 100 bins and then calculate their Pearson correlation to H_{active} counts across the same 100 bins. H_{active} counts indicate the actual active regions. Table 4.3 reports the correlation coefficients between H_{prom} weights and read coverage of H_{active}. We observe that attention weights from our models consistently achieve the highest correlation with the actual active regions near the gene, indicating that this method can capture the important signals for predicting gene activity. Interestingly, we observe that the saliency map on the DeepChrome achieves a higher correlation with H_{active} than the Class-based optimization method for two cell types: H1-hESC (stem cell) and K562 (leukemia cell).

Next, we obtain the attention weights learned by AttentionChrome, representing the important bins and HMs for each prediction of a particular gene as ON or OFF. For a specific gene sample, we can visualize and inspect the bin-level and HM-level attention vectors α_t^j and β^j generated by AttentionChrome. In

Figure 4.3(a), we plot the average bin-level attention weights for each HM for cell type GM12878 (blood cell) by averaging α-maps of all predicted "ON" genes (top) and "OFF" genes (bottom). We see that on average for "ON" genes, the attention profiles near the TSS region are well defined for H_{prom}, H_{enhc}, and H_{struct}. On the contrary, the weights are low and close to uniform for H_{reprA} and H_{reprB}. This average trend reverses for "OFF" genes in which H_{reprA} and H_{reprB} seem to gain more importance over H_{prom}, H_{enhc}, and H_{struct}. These observations make sense biologically as H_{prom}, H_{enhc}, and H_{struct} are known to encourage gene activation while H_{reprA} and H_{reprB} are known to repress the genes [3]. On average, while H_{prom} is concentrated near the TSS region, other HMs like H_{struct} show a broader distribution away from the TSS. In summary, the importance of each HM and its position varies across different genes. E.g., H_{enhc} can affect a gene from a distant position.

In Figure 4.3(b), we plot the average read coverage of H_{active} (top) for the same 100 bins, that we used for input signals, across all the active genes (gene=ON) for GM12878 cell type. We also plot the bin-level attention weights α_t^j for AttentiveChrome (bottom) averaged over all genes predicted as ON for GM12878. Visually, we can tell that the average H_{prom} profile is similar to H_{active}. This observation makes sense because H_{prom} is related to active regions for "ON" genes. Thus, validating our results from Table 4.3.

Finally in Figure 4.3(c) we demonstrate the advantage of AttentiveChrome over LSTM-α model by printing out the β^j weights for genes with differential expressions across the three cell types. That is, we select genes with varying ON(+1)/OFF(−1) states across the three chosen cell types using a heatmap. Figure 4.3(c) visualizes the β^j weights for a certain differentially regulated gene, PAX5. PAX5 is critical for the gene regulation when stem cells convert to blood cells ([85]). This gene is OFF in the H1-hESC cell stage (left column) but turns ON when the cell develops into GM12878 cell (middle column). The β^j weight of repressor mark H_{reprA} is high when gene=OFF in H1-hESC (left column). This same weight decreases when gene=ON in GM12878 (middle column). In contrast, the β^j weight of the promoter mark H_{prom} increases from H1-hESC (left column) to GM12878 (middle column). These trends have been observed in [85] showing that PAX5 relates to the conversion of chromatin states: from a repressive state (H_{prom}(H3K4me3):−, H_{reprA}(H3K27me3):+) to an active state (H_{prom}(H3K4me3):+, H_{reprA}(H3K27me3):−). This example shows that our β^j weights visualize how different HMs work together to influence a gene's state (ON/OFF). We would like to emphasize that the attention weights on both bin-level (α-map) and HM-level (β-map) are gene (i.e. sample) specific.

[3]The small dips at the TSS in both subfigures of Figure 4.3(a) are caused by missing signals at the TSS due to the inherent nature of the sequencing experiments.

The proposed AttentiveChrome model provides an opportunity for a plethora of downstream analyses that can help us understand the epigenomic mechanisms better. Besides, relevant datasets are big and noisy. A predictive model that automatically selects and visualizes essential features can significantly reduce the potential manual costs. The histone code hypothesis states that different histone modifications (HMs) serve as marks for the recruitment of various proteins or protein complexes to regulate diverse chromatin functions, such as gene expression, DNA replication, and chromosome segregation [86]. Biologically, there is evidence for both causal and correlation relationship between HMs and gene expression. It is essential to keep in mind that correlation does not imply causation. A statistical association between an HM and gene expression may simply reflect the fact that they are both related to a third, unknown factor thus the HM may not be regulating the gene expression directly. For example, histone acetylation has also been proposed to physically alter chromatin structure by neutralizing the positive charge of lysines and disrupting intra and internucleosomal interactions, which lead to an open chromatin environment permissible to transcription (**causation**). On the other hand, histone acetylation is also involved in the direct recruitment of proteins that control transcription regulation. Hence, this mark is also highly **correlated** with gene expression while not directly controlling it [87].

Computationally, DeepChrome and AttentiveChrome capture a partial correlation type relationship between a single HM and gene expression. Partial correlation measures the strength of a relationship between two variables - an HM and gene expression in this case - while controlling for the effect of one or more other variables (other HMs). Our models capture a similar relationship between the HM and gene expression. However, we would like to emphasize that while partial correlation, as a term, is associated with linear relationships, we are capturing this relationship ideologically under non-linear settings. Therefore, by using non-linearity and including the combinatorial interactions between different HMs, DeepChrome and AttentiveChrome are not restricted to modeling simple correlations between an HM and gene expression. In fact, they incorporate more information in the form of the effects of other HMs on the gene expression thus providing improved modeling of the underlying mechanism.

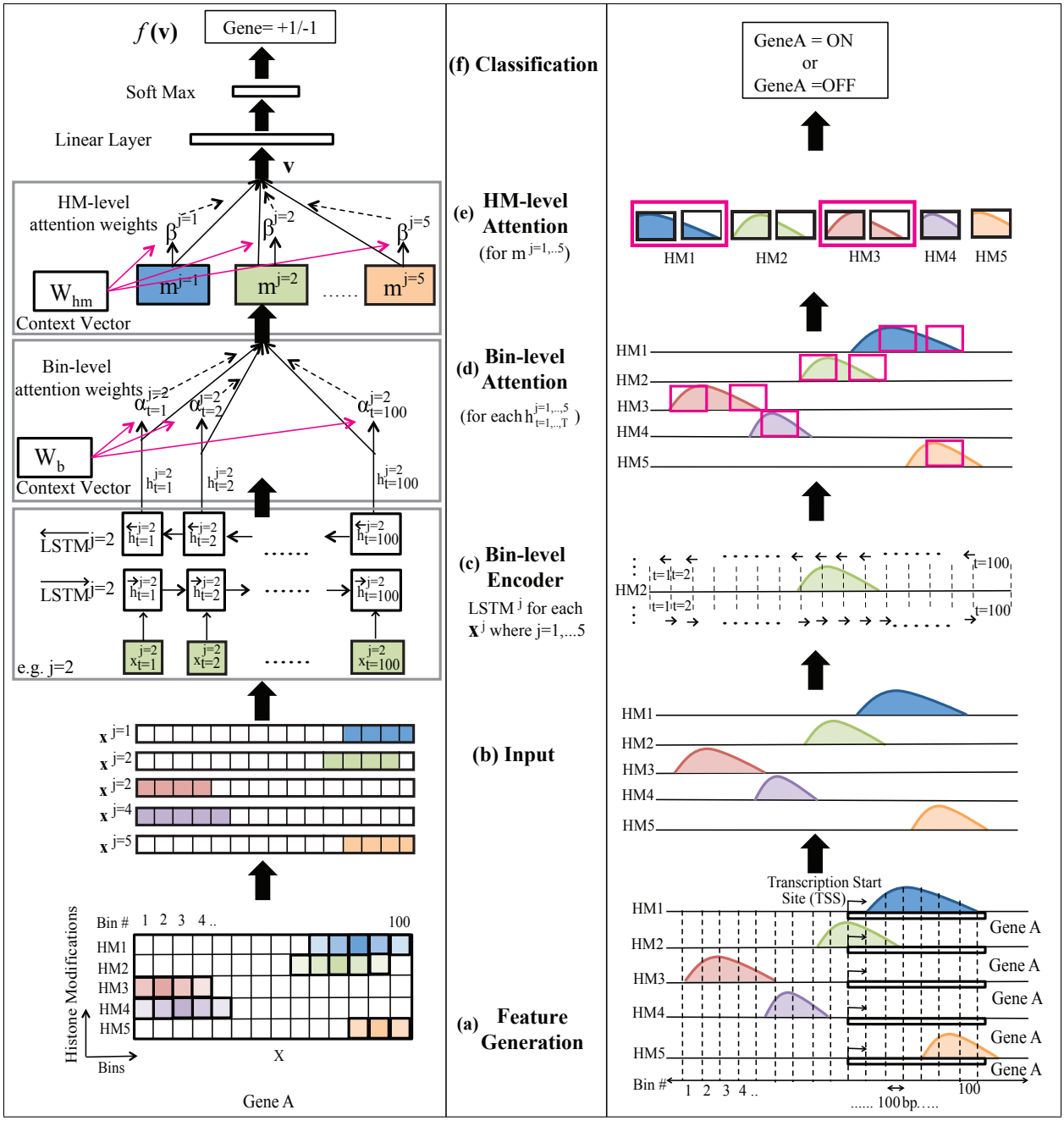

Figure 4.4: Detailed schematic of the proposed AttentiveChrome architecture

AttentiveChrome is a unified framework that can both predict and understand how histone modifications regulate gene expression. We present six steps in order: (a) We generate an input matrix $\mathbf{X}$ for each gene's TSS flanking region, consisting of 100 bins as rows and 5 histone modification (HM) signals as columns. (b) We split the matrix into five vectors representing each HM mark. We input these vectors into the AttentiveChrome model. (c) We use a separate LSTM to learn feature representations of an HM mark. (d) A bin-level attention layer is learned to extract bins that are important for representing an HM mark. This attention layer will aggregate important bins to form an embedding vector for an HM. Here we only show the case of HM2 in steps (c) and (d). (e) Next, to capture the dependencies among different HM marks, we apply another LSTM layer over the representation of 5 HMs. (f) To reward HM marks that are significant clues for classifying an individual gene's expression, AttentiveChrome adds another attention layer- HM-level attention. This layer outputs an embedding vector $\mathbf{v}$ for the whole gene region under consideration. (g) Finally, the output embedding $\mathbf{v}$ from the previous layers will be fed into a classification module to predict the gene expression as high(+1)/low(-1).

Chapter 5

Towards Faster String Kernel Calculation

5.1 Biological Sequence Classification Tasks

Studying molecular sequences gives us deeper insight into the biological processes that can, in turn, help us understand cell development and diseases. Two major tasks essential in the field are Transcription Factor Binding Site (TFBS) Prediction and Remote Protein Homology Prediction.

Transcription factors (TFs) are regulatory proteins that bind to functional sites of DNA to control the regulation of genes. Each different TF binds to specific locations (or sites) on a genomic sequence to regulate cell machinery. Owing to the development of chromatin immunoprecipitation and massively parallel DNA sequencing (ChIP-seq) technologies [88], maps of genome-wide binding sites are currently available for multiple TFs across different organisms. Because ChIP-seq experiments are slow and expensive, computational methods to identify TFBSs accurately are essential for understanding the regulatory functioning and evolution of genomes.

Remote Protein Homology Prediction, i.e., classification of protein sequences according to their biological function or structure, plays an essential role in drug development. Protein sequences that are a part of the same protein superfamily are evolutionally related and functionally and structurally relevant to each other

[89]. Protein sequences with feature patterns showing high homology are classified into the same category. Once classified precisely into a family, the properties of the protein can be easily narrowed down by analyzing only the superfamily it belongs to.

Both these tasks have been formulated as classification tasks, where knowing a DNA or protein sequence, we would like to classify it as a binding site or non-binding site for TF prediction and belonging or not belonging to a protein family for homology prediction.

The state-of-the-art DNN models have been discussed in Section 2.2.1. DNN models give high accuracy when trained on a large number of samples. However, they usually require a lot of parameters which tend to overfit smaller datasets. Also, they use Stochastic Gradient Descent (SGD) for training, which often fails to converge properly for these small size datasets (< 5000 sequences). In many sequence-based applications, including medical research, we can obtain a small number of labeled samples that cannot be trained properly using DNNs. Also, training DNNs can be very time consuming, which becomes an obstacle when handling time-sensitive new experiments in biology.

5.2 Approach

We propose a **fast** algorithmic strategy, GaKCo: **Ga**pped k-mer **K**ernel using **Co**unting to speed up the gapped k-mer kernel calculation. GaKCo uses a "sort and count" approach to calculate kernel similarity through cumulative k-mer counting [12]. GaKCo groups the counting of co-occurrence of substrings at each fixed number of mismatches ($\{0, \ldots, M\}$) into an independent procedure. Such grouping significantly reduces the number of updates on the kernel matrix (an operation that dominates the time cost). This algorithm is naturally parallelizable; therefore we present a multithread variation as our ultimate tool that improves the calculation speed even further.

We provide a rigorous theoretical analysis showing that GaKCo has a better asymptotic kernel computation time cost than gkm-SVM. Our empirical experiments, on three different real-world sequence classification domains, validate our theoretical analysis. For example, for the protein classification task mentioned above where gkm-SVM took more than 5 hours, GaKCo takes only 4 minutes. Compared to GaKCo, gkm-SVM slows down considerably especially when $M \geq 4$ and for tasks with $\Sigma \geq 4$. Experimentally, GaKCo provides a speedup by factors of 2, 100 and 4 for sequence classification on DNA (5 datasets), protein (12 datasets) and

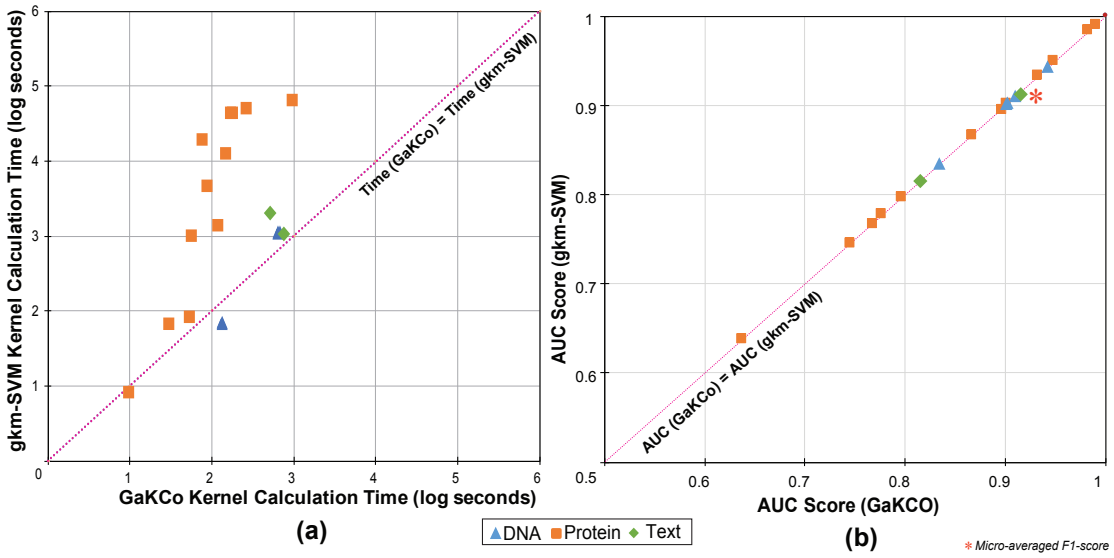

Figure 5.1: GaKCo versus gkm-SVM comparison

(a) Kernel calculation times (log(seconds)) of GaKCo (X-axis) versus gkm-SVM (Y-axis) for 19 different datasets - protein (12), DNA (5), and text (2). GaKCo is faster than gkm-SVM for 16/19 datasets. (b) Empirical performance for the same 19 datasets (DNA, protein, and text) of GaKCo (X-axis) versus gkm-SVM (Y-axis). GaKCo achieves the same AUC-scores as gkm-SVM.

text (2 datasets), respectively, while achieving the same accuracy as gkm-SVM. Figure 5.1(a) compares the kernel calculation times of GaKCo (X-axis) with gkm-SVM (Y-axis). We plot the kernel calculation times for the best performing (g, k) parameters (see Table 5.4) for 19 different datasets. We see that GaKCo is faster than gkm-SVM for 16 out of 19 datasets that we have tested. Similarly, we plot the empirical performance (AUC scores or F1-score) of GaKCo (horizontal axis) versus gkm-SVM (vertical axis) for the best performing (g, k) parameters (see Table 5.4) for the 19 different datasets in Figure 5.1(b). It shows that the empirical performance of GaKCo is same as gkm-SVM with respect to the AUC scores. Table 5.1 summarizes the important notations we use.

5.2.1 Background: Gapped k-mer String Kernels

Spectrum kernel and its mismatch variations generate extremely sparse feature vectors for even moderately sized values of k, since the size of Γ_k is Σ^k. To solve this issue, Ghandi et al. [90] introduced a new set of feature representations, called *gapped k-mers*. It is characterized by two parameters: (1) g, the size of a substring with gaps (we call this gapped instance as g-mer hereafter) and (2) k, the size of non-gapped substring in a g-mer (we call it k-mer). The number of gaps is $(g - k)$. The inner product to compute

Table 5.1: List of symbols and their descriptions that are used.

Notations	Descriptions
D	Dataset under consideration, $D = \{x_1, x_2, \ldots, x_N\}$
N	Number of sequences in a given dataset D
x, x'	Pair of strings in D that are compared for kernel calculation
$K(x, x')$	Kernel Function; equation 5.4 is for the gapped k-mer case
$\phi(x)$	Feature space representation of the string x
l	Average length of sequences in a given dataset D
Σ	Size of the dictionary of a given dataset D
g	Length of the gapped instance or g-mer (specified by the user)
k	Length of k-mer inside a gapped instance (specified by the user)
M	$M = (g - k)$; maximum number of mismatches allowed between two g-mers;
m	Number of mismatches between two g-mers. $m \in \{0, \ldots M\}$
c_{gk}	$c_{gk} = \sum_{m=0}^{M=(g-k)} \binom{g}{m}$.
u	Number of unique g-mers in a given dataset D
z	Number of unique g-mers with > 1 occurrence in a given dataset D
$\mathbf{N}_m(x, x')$	Mismatch profile: number of matching g-mer pairs between x and x' when allowing m mismatches; see equation 5.6
$\mathbf{C}_m(x, x')$	Cumulative mismatch profile: number of matching $\{g - m\}$-mer pairs between x and x'. Each $\{g - m\}$-mer is generated from a g-mer by removing characters from a total of m different positions; See equation 5.5
η	Average size of the *nodelist* of leafnodes in gkm-SVM's trie. Each leafnode is a unique g-mer whose *nodelist* includes all g-mers in the trie whose hamming distance to this leaf is up to M; See equation 5.8

the gapped k-mer kernel function includes sum over all possible k-mer feature counts obtained from the g-mers:

$$K(x, x') = \sum_{\gamma \in \Theta_g} c_x(\gamma) \cdot c_{x'}(\gamma) \tag{5.1}$$

where γ represents a k-mer, Θ_g is the set of all possible gapped k-mers that can appear in all the g-mers (each with $(g - k)$ gaps) in a given dataset (denoted as D hereafter) of sequence samples.

The advantage of this formulation is that it reduces the number of possible k-mers drastically. In a "naive" design of gapped k-mer string kernel when selecting k positions (k-mers) from a g-mer, there can be Σ possible choices for each of the $\binom{g}{k}$ position. Therefore, the total number of possible gapped k-mers equals $F = \binom{g}{k}\Sigma^k$. This feature space grows rapidly with Σ or k. In contrast, Eq. (5.1) (implemented as gkm-SVM [17]) includes only those k-mers whose gapped formulation has appeared as g-mers in a given dataset D. Θ_g includes all unique g-mers of the dataset D, whose size $|\Theta_g|$ is normally much smaller than F because the new feature space is restricted to only observable gapped k-mers in D. Ghandi et al. [17] use this intuition

to reformulate Eq.(5.1) into:

$$K(x, x') = \sum_{i=0}^{l_1} \sum_{j=0}^{l_2} h_{gk}(g_i^x, g_j^{x'}) \tag{5.2}$$

For two sequences x and x' of lengths l_1 and l_2 respectively. g_i^x and $g_j^{x'}$ are the i^{th} and $j^{th}g$-mers of sequences x and x' (i.e., g_i^x is a continuous substring of x starting from the i-th position and ending at the $(i+g-1)^{th}$ position of x). $h_{g,k}$ represents the inner product (or similarity) between g_i^x and $g_j^{x'}$ using the co-occurrence of gapped k-mers as features. $h_{gk}(g_i^x, g_j^{x'})$ is non-zero only when g_i^x and $g_j^{x'}$ have common k-mers. $\textbf{g-pair}_m(x, x')$ denotes a pair of g-mers $(g_1^x, g_2^{x'})$ whose hamming distance is exactly m. g_1^x is from sequence x and $g_2^{x'}$ is from sequence x'.

Each $\textbf{g-pair}_m(.)$ has $\binom{g-m}{k}$ common k-mers, therefore its h_{gk} can be directly calculated as $h_{gk}(\textbf{g-pair}_m) = \binom{g-m}{k}$. Ghandi et al. [17] formulate this observation formally into the coefficient h_m:

$$h_m = \begin{cases} \binom{g-m}{k}, & \text{if } g-m \geq k \\ 0, & \text{otherwise.} \end{cases} \tag{5.3}$$

h_m describes the co-occurrence count of common k-mers for each possible $\textbf{g-pair}_m(.)$ in D. $h_m > 0$ only for cases of $m \leq (g-k)$ or $(g-m) \geq k$. This is because there will be no common k-mers when the number of mismatches (m) between two g-mers is more than $(g-k)$. Now we can reformulate Eq. 5.2 by grouping $\textbf{g-pairs}_m(x, x')$ with respect to different values of m. This is because $\textbf{g-pairs}_m(.)$ with same m contribute the same number of co-occurrence counts: h_m. Thus, Eq. 5.2 can be adapted into the following compact form:

$$K(x, x') = \sum_{m=0}^{g-k} N_m(x, x') h_m \tag{5.4}$$

$N_m(x, x')$ represents the number of $\textbf{g-pair}_m(x, x')$ between sequence x and x'. $N_m(x, x')$ is named as *mismatch profile* by [17]. Now, to compute kernel function $K(x, x')$ for gapped k-mer SK, we only need to calculate $N_m(x, x')$ for $m \in \{0, \ldots g-k\}$, since h_m can be precomputed. The state-of-the-art tool gkm-SVM [17] calculates $N_m(x, x')$ using a trie based data structure that is similar to [11] (with some modifications, details in Section 5.2.3).

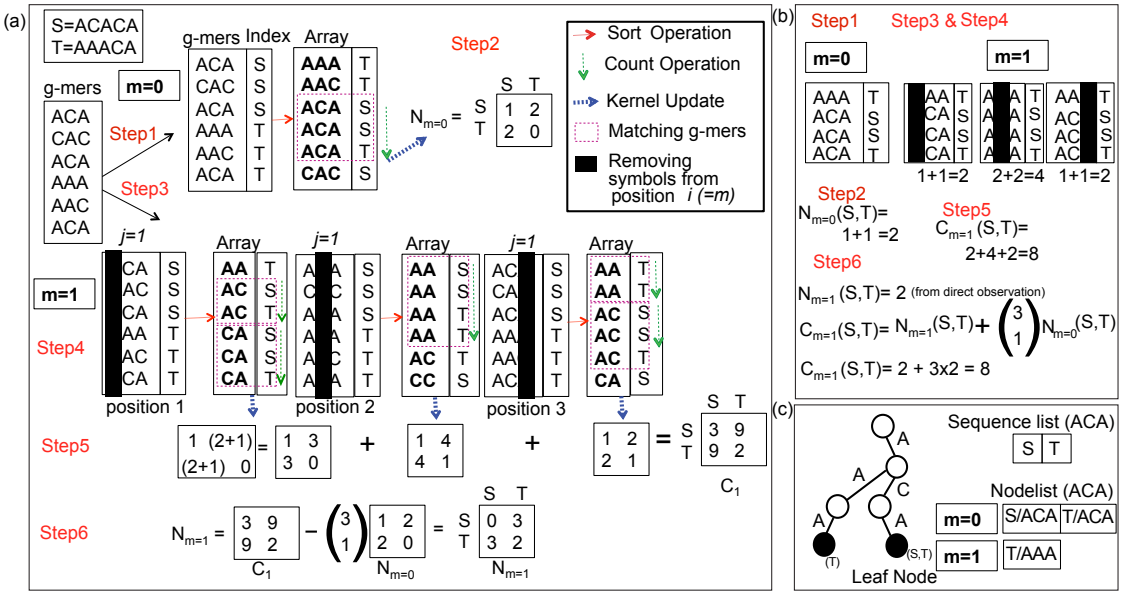

Figure 5.2: Overview of GaKCo algorithm for calculating mismatch profile $N_m(S,T)$.

Given two strings $S = ACACA$ and $T = AAACA$, and $g = 3$, we can get g-mers $\{ACA, CAC, ACA\}$ and $\{AAA, AAC, ACA\}$ respectively. [Step 1] For $m = 0$, all the g-mers are sorted lexicographically. [Step 2] $N_{m=0}(S,T)$ is calculated directly by using sorting and counting to get the counts of matching g-mer ACA in each string $(S/2, T/1)$. Then the kernel update operation updates $N_{m=0}(S,T)$ value to 2. [Step 3] For $m = 1$, we perform over counting of the $g-1$-mers by picking 1 position at a time (from $\binom{g=3}{1}$ positions) and removing symbols from each of these positions to obtain $(g-1)$-mers. [Step 4] We sort and count to find the number of matching $(g-1)$-mers for each picked position. [Step 5] Summing up over all $\binom{g=3}{1}$ positions, we get *cumulative mismatch profile* $C_{m=1}$. $C_{m=1}$ includes matching statistics of g-mers with both mismatches $m = 1$ and $m = 0$. [Step 6] Using Eq. 5.6 we get $N_{m=1}(S,T) = 3$ from $C_{m=1}(S,T) = 9$ and $N_{m=0}(S,T) = 2$. This count corresponds exactly to the actual number of pairs of g-mers at hamming distance $m = 1$ between s and t (i.e. $\{ACA : s/2, AAA : t/1\}, \{CAC : s/1, AAC : t/1\}$). [Note: we do not calculate $N_m(.)$ between a string and itself, i.e $N_m(S,S)$ or $N_m(T,T)$] (b) A case demonstration of the overcounting that takes place when calculating $C_{m=1}$. (c) A case demonstration of two leafnode g-mers and associated nodelist for leaf $\{ACA\}$ in the trie used by gkm-SVM.

5.2.2 Proposed Method: Gapped k-mer Kernel with Counting (GaKCo)

In this chapter, we propose GaKCo, a fast and novel algorithm for calculating gapped k-mer string kernel. GaKCo provides superior time performance over the state-of-the-art gkm-SVM and is different from it in three aspects:

- **Data Structure.** gkm-SVM uses a trie based data structure (plus a separate nodelist at each leafnode) for calculating N_m (see Figure 5.2(c)). In contrast, GaKCo uses simple associative arrays.
- **Algorithm.** GaKCo performs g-mer based cumulative counting of co-occurrence to calculate N_m.
- **Parallelization.** GaKCo groups computations for each value of m into an independent function, making it naturally parallelizable. We, therefore, provide a parallel version that uses multi-thread

implementation.

Intuition:

When calculating N_m between all pairs of sequences in D for each value of m ($m \in \{0, \ldots, M = g - k\}$), we can use counting to process all **g-pairs**$_m(.)$ (details below) from D together. Then we can calculate N_m from such count statistics of **g-pairs**$_m(.)$. This method is entirely different from gkm-SVM that uses a trie to organize g-mers such that each leafnode's (a unique g-mer's) nodelist memorizes its mismatched g-mer neighbors in D for up to $g - k$ mismatches. Section 5.2.3 provides theoretical analysis that GaKCo formulation is asymptotically more scalable to M and Σ than gkm-SVM.

Algorithm:

GaKCo calculates $N_m(x, x')$ as follows (for pseudo code, see Algorithm 2):

1. GaKCo first extracts all possible g-mers from all the sequences in D and puts them in a simple array. Given that there are N number of sequences with average length l [1], the total number of g-mers is $N \times (l - g + 1) \sim Nl$ (see Figure 5.2 (a)).

2. $N_{m=0}(x, x')$ represents the number of **g-pair**$_{m=0}(x, x')$ (pairs of g-mers whose hamming distance is 0) between x and x'. To compute $N_{m=0}(x_i, x_j)$ $\forall i, \forall j = 1, ..., N$, GaKCo sorts all the g-mers lexicographically (see Figure 5.2(a) [Step 1]) and counts the occurrences (if > 1) of each unique g-mer. Then we use these counts and the associated indexes of sequences to update all the kernel entries for sequences that include the matching g-mers (Figure 5.2(a) [Step 2]). This computation is straightforward and the sort and count step takes $O(gNl)$ time cost while the kernel update costs $O(zN^2)$ (at the worst case). Here, z is the number of g-mers that occur > 1 times.

3. For cases when $m = 1, \ldots (g - k)$, we use a statistics measure $C_m(x, x')$, called *cumulative mismatch profile* between two sequences x and x'. This measure describes the number of matching $(g - m)$-mers between x and x'. Each $(g - m)$-mer is generated from a g-mer by removing a total number of m positions. We can calculate the exact *mismatch profile* N_m from the *cumulative mismatch profile* C_m for $m > 0$ (explanation in the next step).

[1] This is a simplification of real world datasets in which sequence length varies across samples

$C_{m=1}$ can be calculated from the associative-array (containing all g-mers in D and their counting statistic) that we obtain from calculating $N_{m=0}$. When $m = 1$, we perform the following operation on the list of all g-mers: we first pick 1 position and remove the symbol from the same position for all g-mers to get a new list of $(g-1)$-mers (Figure 5.2 (a) [Step 3]). We then sort and count this new list to get the number of matching $(g-1)$-mers (Figure 5.2 (b) [Step 4]). For the sequences that have the matching $(g-1)$-mers, we add the counts into their corresponding entries in matrix C_m. This sequence of operations is repeated for a total of $\binom{g}{1}$ positions, i.e every position that can be removed from g-mers to get $(g-1)$-mers. The *cumulative mismatch profile* $C_{m=1}$ is equal to the sum of all counts from all $\binom{g}{1}$ runs (Figure 5.2 [Step 5]). We use the same procedure for calculating C_m for $m = 2, ..., M = g - k$.

4. We now calculate N_m from C_m and N_j for $j = 0, ..., m-1$. First, we explain the relationship between C_m and N_m.

   Given two g-mers g_1 and g_2, we remove symbols from the same set of m positions of both g-mers to get two $(g-m)$-mers: g_1' and g_2'. If the hamming distance between g_1' and g_2': $d(g_1', g_2') = 0$, then we can conclude that the hamming distance between the original two g-mers g_1 and g_2: $d(g_1, g_2) \leq m$ (See formal proof in Appendix). For instance, $C_{m=1}$(x,x') records the statistic of matching $(g-1)$-mers among x and x'. It not only includes the matching statistics of all g-mer pairs whose hamming distance is $m = 1$, but it also over-counts the matching statistics of all g-mer pairs whose hamming distance is $m = 0$. This is because the matching g-mers that were counted for $m = 0$ will also contribute to the matching statistics when considering $(g-1)$-mers and that too for $\binom{g}{1}$ times! Similarly, this over-counting occurs for other values of m as well. Essentially the *cumulative mismatch profile* C_m can be formulated as: $\forall m \in \{0, \ldots, g-k\}$

$$C_m = N_m + \sum_{j=0}^{m-1} \binom{g-j}{m-j} N_j \tag{5.5}$$

   We demonstrate this over-counting using Figure 5.2(b) on a subset of g-mers (ACA,AAA) from Figure 5.2(a). Using Eq.5.5, the exact mismatch profile N_m can be computed as follows:

$$N_m = C_m - \sum_{j=0}^{m-1} \binom{g-j}{m-j} N_j \tag{5.6}$$

   Here, we subtract N_j (for $j = 0, \ldots, i-1$) from C_m to compensate for the over-counting described above.

Algorithm 2 GaKCo

Require: L, g, k ▷ L=Array list of g-mers

procedure CalculateKernel(L,g,k)
  $M \leftarrow g - k$
  N←MismatchProfile(L,g,M)
  $K \leftarrow 0$
  for $m : 0 \rightarrow M$ **do**
    $h_m \leftarrow \binom{g-M}{k}$
    $K \leftarrow K + N_m.h_m$
procedure MismatchProfile(L,g,M)
  for $m : 0 \rightarrow M$ **do** ▷ Parallel threads
    $C_m \leftarrow 0$ ▷ Cumulative Profile
    $n_{pos} \leftarrow \binom{g}{m}$ ▷ Number of positions
    for $i : 0 \rightarrow n_{pos}$ **do**
      $C_m^i \leftarrow 0$
      $L^i \leftarrow removePosition(L, i)$
      $L^i \leftarrow sort(L^i)$
      $C_m^i \leftarrow countAndUpdate(L^i)$
      $C_m \leftarrow C_m + C_m^i$
  for $m : 0 \rightarrow M$ **do**
    for $j : 0 \rightarrow m - 1$ **do**
      $N_m \leftarrow C_m - \binom{g-j}{m-j} N_m$
  return N ▷ $\mathbf{N} \in \{N_0, \ldots, N_M\}$

Ensure: K ▷ Kernel Matrix

5.2.3 Theoretical Comparison of Time Complexity

In this section, we conduct asymptotic analysis to compare the time complexities of GaKCo with the state-of-the-art toolbox gkm-SVM.

Time Complexity of GaKCo:

The time cost of GaKCo splits into two groups: (1) Pre-processing: those operations that indirectly update the matching statistics among sequences; (2) Kernel updates: those operations that directly update the matching statistics among sequences.

Pre-processing: For each possible m ($m \in \{0, \dots M = g-k\}$), GaKCo needs to choose m positions for symbol removing (Figure 5.2 (a) [Step 3]), and then sort and count the possible $(g-m)$-mers from D (Figure 5.2 (a) [Step 4]). Therefore the time cost of pre-processing is $O(\Sigma_{m=0}^{M=g-k}\binom{g}{m}(g-m)Nl) \sim O(\Sigma_{m=0}^{M}\binom{g}{m}gNl)$. To simplifying notations, we use c_{gk} to represent $\sum_{m=0}^{M=(g-k)}\binom{g}{m}$ hereafter.

Kernel Updates: These operations update the entries of C_m or N_m matrices when GaKCo finishes each round of counting the number of matching $(g-m)$-mers. Assuming z denotes the number of unique $(g-m)$-mers that occur > 1 times, the time cost of kernel update operations is (at the worst case) equivalent to $O(\Sigma_{m=0}^{M}\binom{g}{m}zN^2) \sim O(c_{gk}zN^2)$. Therefore, the overall time complexity of GaKCo is $O(C_{gk}[gNl + zN^2])$.

Parallelization:

m-based Parallelization: For each value of m from $\{0, \dots M = g-k\}$, calculating C_m is independent from other values of m. Therefore, GaKCo's algorithm can be easily revised into a parallel version. Essentially, we just need to revise Step 9 in Algorithm 1 (pseudo code) - "For each value of m"- into, "For each value of m per machine/per core/per thread". In our original implementation, we create a thread for each value of m from $\{0, \dots M = g-k\}$ and calculate C_m in parallel. In the end, we compute the final kernel matrix K using all the resulting C_m matrices. Figure 5.4 and 5.7(b) show the improvement of kernel calculation speed when comparing the multi-thread version with the single-thread implementation of GaKCo.

Smarter parallelization strategy (GaKCo2.0): In the original Gakco, we thread over mismatch values m. Thread m computes the cumulative mismatch profile with m mismatches, where $0 \leq m \leq (g-k)$. The problem is that in order to produce the m^{th} cumulative mismatch profile (C_m), we must first perform $\binom{g}{m}$ sort-count-update steps (Figure 5.2 Step 5). So the number of units of work each thread needs to do varies greatly depending on its value of m. For example, when $g = 5$ and $k = 1$. The GaKCo implementation creates 5 threads, each of which needs to perform $\binom{g}{m}$ sort-count-update steps:

Thread 0: $\binom{5}{0}$ = 1 unit of work

Thread 1: $\binom{5}{1}$ = 5 units of work

Thread 2: $\binom{5}{2}$ = 10 units of work

Thread 3: $\binom{5}{3}$ = 10 units of work

Thread 4: $\binom{5}{4}$ = 5 units of work.

Threads 3 and 4 have twice as much work as threads 2 and 5. And threads 3 and 4 have ten times as much work as thread 1. Gakco2.0 improves this implementation by using a smarter paralleization strategy. We first pre-calculate how many sort-count-update steps need to be computed in total (i.e., 31 units of work in the above example) as follows:

$$TotalSteps = \sum_{m=0}^{(g-k)} \binom{g}{m} \tag{5.7}$$

Next, we create a "work queue" of length equal to the total number of steps. Each element of the queue is a small C++ *struct* containing the m and combination number values. Note that each step is associated with a m value and a combination number. Therefore, for a given value of m, we have a sort-count-update step for each of the $\binom{g}{m}$ combinations, and we keep track of that using *struct* in the work queue.

We create threads and divide the work queue among them. Each thread will deduce the sort-count-update step it currently needs to compute by accessing *struct* in the work queue. When it finishes calculating one sort-count-update step and adding the result to the associated cumulative mismatch profile kernel, it moves to the next assigned step. The thread stops on completing all the assigned sort-count-update steps.

Through experimentation, we found that speed and memory usage are best when we create 1 thread for each the machines CPU cores. However, for machines with > 16 cores, we create 20 threads by default. We also provide an option for the user to specify how many threads they would like to create.

Thread Synchronization: Two threads that are performing sort-count-update steps might frequently need to add the results to the same C_m profile kernel. This situation results in a race condition that can be solved using synchronization techniques. We associate each C_m profile profile with a *mutex* lock [2] thus, enforcing synchronization. When a thread calls *lock(mutex)* for mismatch $m = m'$ none of the other threads can touch

[2]We use the C++11 threading library and standard *pthread mutexes*.

that $C_{m'}$ profile kernel during update. They must wait for that thread to call *unlock(mutex)* first. Meanwhile, a thread that needs to update C_m for different $m \neq m'$, may do so. Thus, the synchronization only affects threads that are trying to update the same C_m profile. The strategy has been summarized in pseudo code: Algorithm 3.

Algorithm 3 GaKCo2.0 Parallelization

Require: L, g, k ▷ L=Array list of g-mers

1: $M \leftarrow g - k$
2: **procedure** MismatchProfile(L,g,M)
3: $C_m \leftarrow 0$ ▷ Cumulative Profile
4: TotalSteps $\leftarrow \sum_0^M \binom{g}{m}$ ▷ Construct Work Queue
5: **for** $i : 0 \rightarrow len(WorkQueue)$ **do** ▷ Parallel threads
6: $C_m^i \leftarrow 0$
7: $L^i \leftarrow removePosition(L, i)$
8: $L^i \leftarrow sort(L^i)$
9: $C_m^i \leftarrow countAndUpdate(L^i)$ ▷ lock(mutex)
10: $C_m \leftarrow C_m + C_m^i$ ▷ unlock(mutex)

gkm-SVM Algorithm

Now we introduce the algorithm of gkm-SVM briefly. Given that there are N sequences in a dataset D, gkm-SVM first constructs a trie tree recording all the unique g-mers in D. Each leafnode in the trie stores a unique g-mer (more precisely by its path to the rootnode) of D. We use u to denote the total number of the unique g-mers in this trie. Next, gkm-SVM traverses the tree using the order of depth-first. For each leafnode (like ACA in (Figure 5.2 (c)), it maintains a *nodelist* that includes all those g-mers in D whose hamming distance to the leafnode g-mer $\leq M$. When accessing a leafnode, all mismatch profile matrices $N_m(x, x')$ for $m \in \{0, \ldots, M = (g - k)\}$ are updated for all possible pairs of sequences x and x'. Here x consists of the g-mer of the current leafnode (like S/ACA in (Figure 5.2 (c)). x' belongs to the *nodelist*'s sequence list. x' includes a g-mer whose hamming distance from the leafnode is m (like $T/ACA(m = 0)$ or $T/AAA(m = 1)$ in (Figure 5.2 (c)).

Implementations	GaKCo	gkm-SVM [17]
Pre-processing	$c_{gk}gNl$	ug
Kernel updates	$c_{gk}zN^2$	ηuN^2

Table 5.2: Comparing time complexity of gkm-SVM versus GaKCo.

gvm-SVM's time cost is $O(ug+\eta uN^2)$. GaKCo's time complexity is $O(c_{gk}[gNl + zN^2])$. In gkm-SVM the term ηuN^2 dominates the time. For GaKCo the term $c_{gk}zN^2$ dominates the time cost.

Time Complexity of gkm-SVM:

We also split operations of gkm-SVM into those indirectly (pre-processing) or directly (kernel-update) updating N_m.

Pre-processing: Assuming u unique g-mers exist in D, then the number of leafnodes in the trie is u. The time taken to construct the trie equals $O(ug)$.

Kernel Update: For each leafnode of the trie (total u nodes), for each g-mer in its nodelist (assuming average size of nodelist is η), gkm-SVM uses the matching count among g-mers to update involved sequences' entries in N_m (if hamming distance between two g-mers is m). Therefore the time cost is $O(\eta uN^2)$ (at the worst case). Essentially η represents on average the number of unique g-mers (in the trie) that are at a hamming distance up to M from the current leafnode. That is

$$\eta = min(u, \sum_{m=0}^{M=(g-k)} \binom{g}{m}(\Sigma-1)^m) \sim min(u, c_{gk}(\Sigma-1)^M) \tag{5.8}$$

Figure 5.3 shows that η grows exponentially to M until reaching its maximum u. The total complexity of time cost from gkm-SVM is thus $O(ug + u\eta N^2)$. Asymptotically, at the worst case when $\eta = u$, the time complexity of gkm-SVM is $O(ug + u^2N^2)$.

Comparing Time Complexity of GaKCo with gkm-SVM:

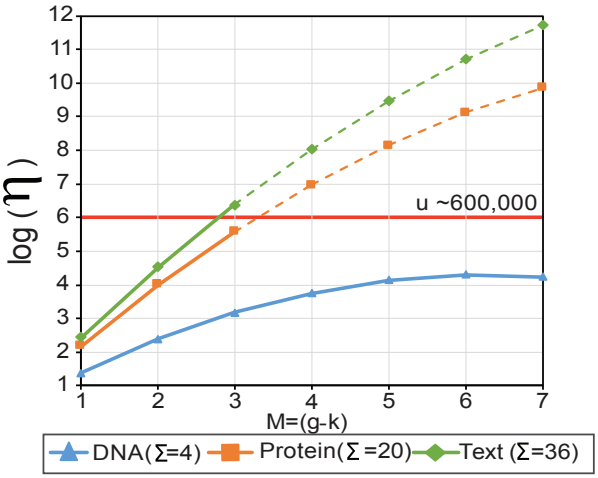

Figure 5.3: Increase in estimated size of *nodelist* with increasing M.

η: estimated size of *nodelist* used in gkm-SVM [17]. It is equal to the number of g-mers with M mismatches from the current g-mer at the trie-leafnode. For a given g-mer g, the number of possible g-mers that are at a distance M is roughly $c_{gk}(\Sigma-1)^M$. The size grows exponentially with the number of mismatches m. When dictionary size Σ is small, $\eta < u$. However when $\Sigma > 4$ and $m \geq 4$, $\eta \leq u$.

Table 5.2 compares the asymptotic time cost of GaKCo with gkm-SVM. Asymptotically the time complexity of gvm-SVM is $O(ug + \eta u N^2)$. For GaKCo the overall time complexity is $O(c_{gk}[gNl + zN^2])$. In gkm-SVM the term $O(\eta u N^2)$ dominates the overall time. For GaKCo the term $O(c_{gk} z N^2)$ dominates the time cost. For simplicity, we assume that $z = u$ even though $z \leq u$. Upon comparing $O(\eta \times uN^2)$ of gkm-SVM with $O(c_{gk} \times uN^2)$ of GaKCo, clearly the difference lies between the terms η in gkm-SVM and c_{gk} in GaKCo. In details,

- $\eta \sim c_{gk}(\Sigma-1)^M$ (gkm-SVM) versus c_{gk} (GaKCo): For a given g-mer g, the number of possible g-mers that are at a distance M from it is $c_{gk}(\Sigma-1)^M$. That is, $\binom{g}{M}$ positions can be substituted with $(\Sigma-1)^M$ possible characters. Thus in gkm-SVM, the estimated size η grows exponentially with number of allowed mismatches M. We show the trend of function $f = c_{gk}(\Sigma-1)^M$ in Figure 5.3 (a) for three different domains - TF-DNA ($\Sigma = 4$), SCOP-protein ($\Sigma = 20$) and text ($\Sigma = 36$) by varying the values of M for $g = 10$. We threshold these curves at $u = 6 \times 10^4$, which is the average observed value of u across multiple datasets. In Figure 5.3 (a), when dictionary size Σ is small (=4), the size of the *nodelist* η is mostly smaller than u. But when Σ is larger than 4, η gets larger than u for $M \geq 4$. In contrast, GaKCo's term c_{gk} is independent of the dictionary size Σ.

- $\eta = u$ (gkm-SVM) versus c_{gk} (GaKCo): For large dictionary size (e.g. $\Sigma = 20$), size of the nodelist η mostly equals to u in gkm-SVM. Even for cases with small dictionary size (e.g. $\Sigma = 4$) η is close to u for $M \geq 4$. While gkm-SVM might be fast for small Σ and $M < 4$, its kernel calculation time will slow down considerably for $M \geq 4$. For example, for one of the SCOP datasets, when $g = 10$, count of unique g-mers $u = 6 \times 10^4$ at $M = 4$ (close to u shown in Figure 5.3 (a)). Therefore, at a modest value of $M = 4$ $\eta = 6 \times 10^6$ for gkm-SVM while $c_{gk} = 210$ for GaKCo. The former is approximately 300 times higher than GaKCo.

5.2.4 Justification of GaKCo's Sort and Count Method

A core piece of the GaKCo's kernel computation is counting the observed g-mers in the strings for which the kernel value is being computed. The final implementation of our algorithm uses a sorting-based counting method, proposed by Kuksa et al. [12] [3], but we did consider a hashing approach as well. There are straightforward time complexity justifications for choosing sorting over hashing, which we explain in this section.

A hash table, treated as an associative array, could easily be used to count instances of a g-mer. Given a g-mer, which consists of a g-length token and a reference to the original string number, we may write a simple hash function that executes in $\Theta(g)$ time (as we ought to consider every character in the string for a well-distributed hash). Also, given that the total number of strings is N of average length l, then the total number of g-mers is $\sim Nl$. If we accept the "typical-case" runtime of insertion into a hash table, which is $\Theta(1)$, then to count every g-mer we must perform at least $\Theta(g \cdot Nl)$ steps: for each of the total Nl g-mers, we do g work to hash, insert, and update the associated value.

At first consideration, a sorting-based approach would seem to be strictly worse, as any swapping sort would take $\Theta(g \cdot (Nl) \lg(Nl))$ time. However, using a non-swapping sort, in our case, gives us $\Theta(g \cdot Nl)$ time, which is the same as we derived for the above hashing method. However, the sorting requires exactly $g \cdot Nl$ steps, while the hashing approach needs more steps to resolve any possible collisions. To confirm our theoretical justification, we implemented hashing approach and found that our sorting method was, indeed, faster than hashing.

[3] [12] first proposed cumulative mismatch profile concept for their mismatch kernel calculation. However, they use all possible k-mers built from the dictionary with m mismatches as the feature space. Thus, the authors [12] need to precompute a complex weight matrix to incorporate all possible k-mers with m mismatches. This computation cost $O((2m+1)g^{(m+1)}(m+2)^m)$ making it exponential in g.

Table 5.3: Details of datasets used for different prediction tasks.

All tasks, except WebKB, are binary classification tasks. WebKB is a multi-class classification dataset with four classes: *project, course, faculty,* and *student*.

			Training		**Testing**		**Sample properties**		
Prediction Task	Repo	Datasets	Pos seq	Neg seq	Pos seq	Neg seq	N	Σ	Max(l)
12cmTF Binding Site(DNA)	ENCODE	CTCF	1000	1000	1000	1000	4000	5	100
		EP300							
		JUND							
		RAD21							
		SIN3A							
12cmRemote Protein Homology(Protein)	SCOP	1.1	1150	1189	8	1227	3574	20	905
		1.34	866	1209	6	1231	3312		
		2.19	110	1235	9	1206	2560		
		2.31	1063	1235	8	1194	3500		
		2.1	4763	1229	120	950	7062		
		2.34	286	1215	6	1231	2738		
		2.41	192	1235	6	1213	2646		
		2.8	56	1185	8	1231	2480		
		3.19	922	1181	7	1231	3341		
		3.25	1187	1208	11	1231	3637		
		3.33	466	1214	7	1231	2918		
		3.50	105	1231	8	1205	2549		
Text Classification	Stanford Treebank	Sentiment	3883	3579	877	878	9217	36	260
	Dataset from [91]	WebKB	335, 620, 744, 1083		166, 306, 371, 538		4163	36	14218

5.3 Experimental Setup

5.3.1 Benchmark Tasks of Sequence Classification

DNA and Protein Sequence Classification

Studying DNA and Protein sequences gives us deeper insight into the biological processes that can, in turn, help us understand cell development and diseases. Two major tasks essential in the field are Transcription Factor Binding Site (TFBS) Prediction and Remote Protein Homology Prediction.

Transcription factors (TFs) are regulatory proteins that bind to functional sites of DNA to control the regulation of genes. Each different TF binds to specific locations (or sites) on a genomic sequence to regulate cell machinery. Owing to the development of chromatin immunoprecipitation and massively parallel DNA sequencing (ChIP-seq) technologies [88], maps of genome-wide binding sites are currently available for multiple TFs across different organisms. Because ChIP-seq experiments are slow and expensive, computational methods to identify TFBSs accurately are essential for understanding cell regulation.

Remote Protein Homology Prediction, i.e., classification of protein sequences according to their biological function or structure, plays a significant role in drug development. Protein sequences that are a part of the same protein superfamily are evolutionally related and functionally and structurally relevant to each other

[89]. Protein sequences with feature patterns showing high homology are classified into the same superfamily. Once assigned a family, the properties of the protein can be easily narrowed down by analyzing only the superfamily to which it belongs.

Researchers have formulated both these tasks as classification tasks, where knowing a DNA or protein sequence, we would like to classify it as a binding site or non-binding site for TF prediction and belonging or not belonging to a protein family for homology prediction respectively.

Text Classification

Text classification incorporates multiple tasks like assigning subject categories or topics to documents, spam detection, language detection, sentiment analysis, etc. Generally, given a document and a fixed number of classes, the classification model has to predict the class that is most relevant to that document. Several recent studies have discovered that character-based representation provides straightforward and powerful models for relation extraction [16], sentiment classification [92], and transition based parsing [93]. Lodhi et al. [94] first used string kernels with character level features for text categorization. However, their kernel computation used dynamic programming which was computationally intensive. Over recent years, more efficient string kernel methods have been devised [11, 12, 17, 52, 95]. Therefore, we use simple character-based text input for document and sentiment classification task.

We perform 19 different classification tasks to evaluate the performance of GaKCo. These tasks belong to the discussed three categories: (1) TF binding site prediction (DNA dataset), (2) Remote Protein Homology prediction (protein dataset), and (3) Character-based English text classification (text dataset).

5.3.2 Experimental Setup

Datasets:

- *ENCODE ChIP-Seq DNA Sequences:* Maps of genome-wide binding sites are currently available for multiple TFs for human genome via the ENCODE [72] database. These ChIP-seq "maps" mark the positions of the TF binding sites. We select 100 basepair sequences overlapping the binding sites as positive sequences and randomly select non-binding sequences from the human genome as negative sequences. We perform this selection for five different transcription factors (CTCF, EP300, JUND,

RAD21, and SIN3A) from the K562 (leukemia) cell type, resulting in five different prediction tasks. We select 2000 sequences for training that consist of 1000 positive and negative samples each. For testing, we use another set of 2000 sequences with 1000 positive and negative samples each. We use the dictionary size of 5 ($\Sigma = 5$). There are four nucleotide symbols - A, T, C, G - in the DNA. Additionally, sometimes sequences have 'N' for nucleotides that are not read by the sequencing machines. Therefore, the dictionary is {A,T,C,G,N}.

- *SCOP Protein Sequences:* The SCOP domain database consists of protein domains, no two of which have 90% or more residual identity [18]. It is hierarchically divided into folds, superfamilies, and finally families. We use 12 sets of samples (listed in Table 5.3) and select positive test sequences (for each sample) from 1 protein family of a particular superfamily. We obtain the positive training sequences from remaining families in that superfamily. We select negative training and test sequences from non-overlapping folds outside the positive sequence fold. We use the dictionary size of 20 ($\Sigma = 20$) as there are 20 amino acid symbols that make up a protein sequence. Therefore the dictionary is {A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}.

- *WebKB and Sentiment Classification Datasets:* The documents in the WebKB are web pages collected by the World Wide Knowledge Base project of the CMU text learning group and were downloaded from The 4 Universities Data Set Homepage. These pages were collected from computer science departments of various universities in 1997. We downloaded the processed datasets (removed stop/short words, stemming, etc.) from [91]. This task is a multi-class classification task with four classes: project, course, faculty, and student. For the sentiment analysis experiments, we used the Stanford sentiment treebank dataset [96]. This dataset provides a score for each sentence between 0 and 1 with $[0, 0.4]$ being negative sentiment and $[0.6, 1.0]$ is positive. We combined the validation set in the original treebank dataset with the training set. We use the dictionary size of 36 ($\Sigma = 36$) since we use character-based input. The dictionary includes all the alphabets [A-Z] and numbers [0-9].
  Details of the datasets are in Table 3.

Baselines: We compare the kernel calculation times and empirical performance of GaKCo with gkm-SVM [17]. We also run the CNN implementation from [8] for all the datasets.

Classification: After calculation, we input the $N \times N$ kernel matrix into an SVM classifier as an empirical feature map using a linear kernel in LIBLINEAR [97]. Here N is the number of sequences in each dataset. The SVM maximizes the margin between the positive and negative instances of the samples in the kernel defined feature space. For the multi-class classification of WebKB data, we use the multi-class version of LIBSVM [98].

Model parameters: We vary the hyperparameters $g \in \{7, 8, 9, 10\}$ and $k \in \{1, 2, \ldots, g-1\}$ of both GaKCo and gkm-SVM. $M = (g-k)$ for all these cases. We also tune the hyperparameter $C \in \{0.01, 0.1, 1, 10, 100, 1000\}$ for the SVM. We present the results for the best g, k, and C values based on the empirical performance metric. We ran the CNN model with default parameters for 50 epochs (number of training and testing times), and we present the results for the epoch with the best empirical performance metric.

Evaluation Metrics:

- *Running time:* We compare the kernel calculation times of GaKCo and gkm-SVM in seconds. In some figures, we have represented time in log-scale (log(seconds)) to accommodate large values. All run-time experiments have been performed on AMD Opteron(TM) Processor 6376 @ 2.30GHz with 250GB memory.
- *Empirical performance:* We use the Area Under Curve (AUC) score (from the Receiver Operating Characteristic (ROC) curve) as our empirical evaluation metric for 18 binary classification tasks. We report the results of WebKB multi-class classification using micro-averaged F1 score.

5.4 Results

5.4.1 Kernel Calculation Time Performance

Our experimental results confirm our theoretical analysis in Section 5.2 that GaKCo has a lower kernel calculation time than gkm-SVM. Figure 5.1(a) shows GaKCo is faster than gkm-SVM for 16/19 tasks. The other three tasks for which GaKCo records similar kernel calculation times are DNA sequence prediction tasks. This is expected as DNA has a smaller dictionary size ($\Sigma = 5$) and thus, for a small number of

allowed mismatches (M) gkm-SVM gives comparable speed performance. We elaborate on this further in the following discussion.

GaKCo scales better than gkm-SVM for large dictionary size (Σ) and large number of mismatches (M):

Figure 5.4 shows the kernel calculation times of GaKCo and gkm-SVM for the best-performing g and varying $k = \{1, 2, \ldots (g-1)\}$ for three binary classification datasets: (a) EP300 (DNA), (b) 1.34 (protein), and (c)Sentiment (text) respectively. We select these three datasets as they achieve the best AUC scores out of all 19 tasks (see Table 5.4). We fix g and vary k to show time performance for different number of allowed mismatches i.e. $M = (g-k) = \{1, 2, \ldots (g-1)\}$. For GaKCo, the results are plotted for both single-thread and the multi-thread implementations. We refer to the multi-thread implementation as GaKCo because that is our final code version. Our results show that GaKCo (single-thread) scales better than gkm-SVM for a large dictionary size (Σ) and a large number of mismatches (M). The final version of GaKCo (multi-thread) further improves the performance. Details for each dataset are as follows:

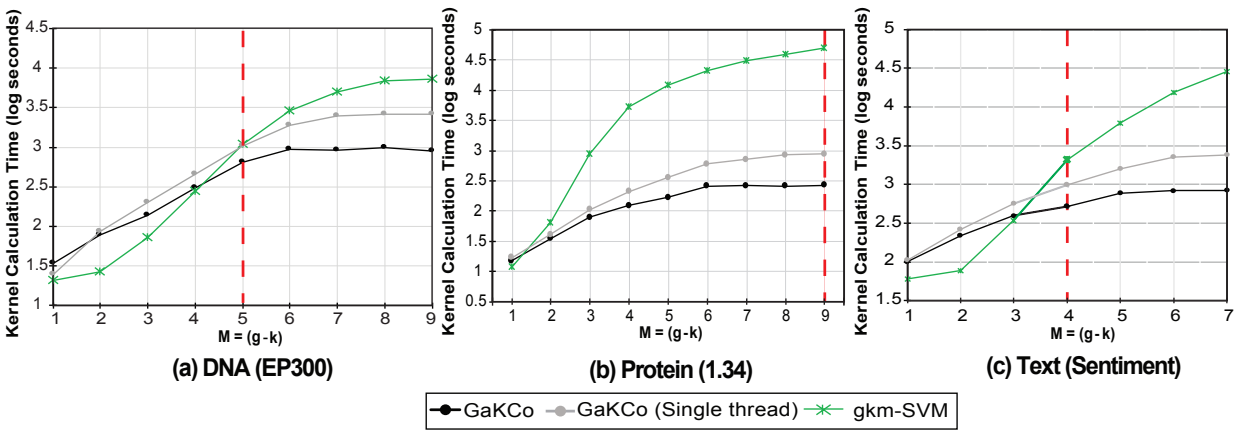

Figure 5.4: Kernel calculation times (lower is better) for best g and varying k with M.

(a) EP300 (DNA,$\Sigma = 5$), (b) 1.34 (protein, $\Sigma = 20$), and (c) Sentiment (text, $\Sigma = 36$) datasets. The best performing hyperparameters (g, k or $M = (g-k)$) are highlighted as red colored dashed lines. For GaKCo, results are shown for both single thread and multi-thread implementations. GaKCo (single thread) outperforms gkm-SVM for a large dictionary size ($\Sigma > 5$) and a large number of mismatches $M \geq 4$, confirming our analysis in Section 5.2. The final GaKCo (multi-thread) implementation further improves the performance. For protein dataset (b) gkm-SVM takes > 5 hours to calculate the kernel, while GaKCo calculates it in 4 minutes.

- DNA dataset ($\Sigma = 5$): In Figure 5.4 (a), we plot the kernel calculation times for best $g = 10$ and varying k with $M = \{1, 2, \ldots 9\}$ for EP300 dataset. As expected, since the dictionary size of DNA dataset (Σ) is small, gkm-SVM performs fast kernel calculations for $M = (g-k) < 4$. However, for large $M \geq 4$,

its kernel calculation time increases considerably compared to GaKCo. This result connects to Figure 5.3 in Section 5.2, where our analysis showed that the *nodelist* size becomes closer to u as M increases, thus increasing the time cost.

- Protein dataset ($\Sigma = 20$): Figure 5.4 (b), shows the kernel calculation times for best $g = 10$ and varying k with $M = (g - k) = \{1, 2, \ldots 9\}$ for 1.34 dataset. Since the dictionary size of protein dataset (Σ) is larger than DNA, gkm-SVM's kernel calculation time is worse than GaKCo even for smaller values of $M < 4$. This also connects to Figure 5.3 where the size of *nodelist* $\sim u$ even for small M for protein dataset, resulting in higher time cost. For best-performing parameters $g = 10, k = 1(M = 9)$, gkm-SVM takes 5 hours to calculate the kernel, while GaKCo calculates it in 4 minutes.

- Text dataset ($\Sigma = 36$): Figure 5.4 (c), shows the kernel calculation times for best $g = 8$ and varying k with $M = \{1, 2, \ldots 7\}$ for Sentiment dataset. The results follow the same trend as presented above. For large $M \geq 4$, kernel calculation time of gkm-SVM is slower as compared to GaKCo. One would expect that with large dictionary size (Σ) the performance difference will be same as that for protein dataset. However, unlike protein sequences, where the substitution of all 20 characters in a g-mer is equally likely, text dataset has an underlying structure. Concretely, the chance of substitution of some characters in a g-mer will be higher than others. For example, in a given g-mer "my nam", the last position is more likely to be occupied by 'e' than 'z'. Therefore, even though the dictionary size is large, the growth of the *nodelist* is restricted by the underlying structure of the text. Therefore, while GaKCo's time performance is consistent across all three datasets, gkm-SVM's time performance varies due to the characteristic properties (like dictionary size (Σ)) of the datasets.

According to our asymptotic analysis in Section 5.2, GaKCo should always be faster than gkm-SVM. However, in Figure 5.4 we notice that for certain cases (e.g. for DNA when $M < 4$ in Figure 5.4) GaKCo's speed is lower than gkm-SVM. This is because, in our analysis, we theoretically estimate the size of gkm-SVM's *nodelist*. We see that in practice, the actual *nodelist* size is smaller than our estimate for certain cases where gkm-SVM is faster than GaKCo. However, with a larger value of $M(\geq 4)$ or dictionary size ($\Sigma > 5$), the nodelist size in practice matches our theoretical estimation. Therefore, GaKCo always has lower kernel calculation times than gkm-SVM for these cases.

GaKCo is independent of dictionary size (Σ):

GaKCo's time complexity analysis (Section 5.2) shows that it is independent of the Σ^M term, which controls the size of gkm-SVM's *nodelist*. In Figure 5.6 (a), we plot the average kernel calculation times for the best performing (g, k) parameters for DNA ($\Sigma = 5$), protein ($\Sigma = 20$), and text ($\Sigma = 36$) datasets respectively. The results validate our analysis. We find that gkm-SVM takes similar time as GaKCo to calculate the kernel for DNA dataset due to the small dictionary size. However, when the dictionary size increases for protein and text datasets, it slows down considerably. GaKCo, on the other hand, is consistently faster for all three datasets, despite the increase in dictionary size.

GaKCo algorithm benefits from parallelization:

m-based parallelization: As discussed earlier, the calculation of C_m (such that $m = \{0, 1 \dots M = (g - k)\}$) is an independent procedure in GaKCo's algorithm. This property makes GaKCo naturally parallelizable. We first implement the parallelized version of GaKCo by distributing calculation of C_m across m threads where, $m = \{0, 1 \dots M = (g - k)\}$. We have already witnessed that the multi-thread version of GaKCo improves the speed of its single thread version in Figure 5.4. Next, in Figure 5.6(b) we plot the average kernel calculation times across DNA (5), protein (12) and text (2) datasets for both multi-thread and single thread implementations. Through this figure, we demonstrate that the improvement in speed by parallelization is consistent across all datasets.

Smarter parallelization (GaKCo2.0): Next, we demonstrate the improvement over original GaKCo m-based multi-threading by adopting a smarter parallelization strategy that we call GaKCo2.0. Figure 5.5 shows the kernel calculation profiles of original GaKCo versus GaKCo2.0 for varying values of mismatches $M = (g - k) = \{1, \dots, 9\}$ and both (a) DNA dataset (EP300) and (b) Protein dataset (1.34). It is evident that GaKCo2.0 is consistently faster than the original GaKCo multi-thread version that performed uneven splitting of work across threads. We also compare both the GaKCo implementations with the gkm-SVM and introduce results from its multi-thread version (gkm-SVM-2.0) [99] and show that overall GaKCo2.0 performs the fastest for larger values of mismatches (i.e $M > 3$). Since both GaKCo2.0 and gkm-SVM-2.0 allow the users to define the number of threads, we set them to 20. Original GaKCo, however, uses M number of threads.

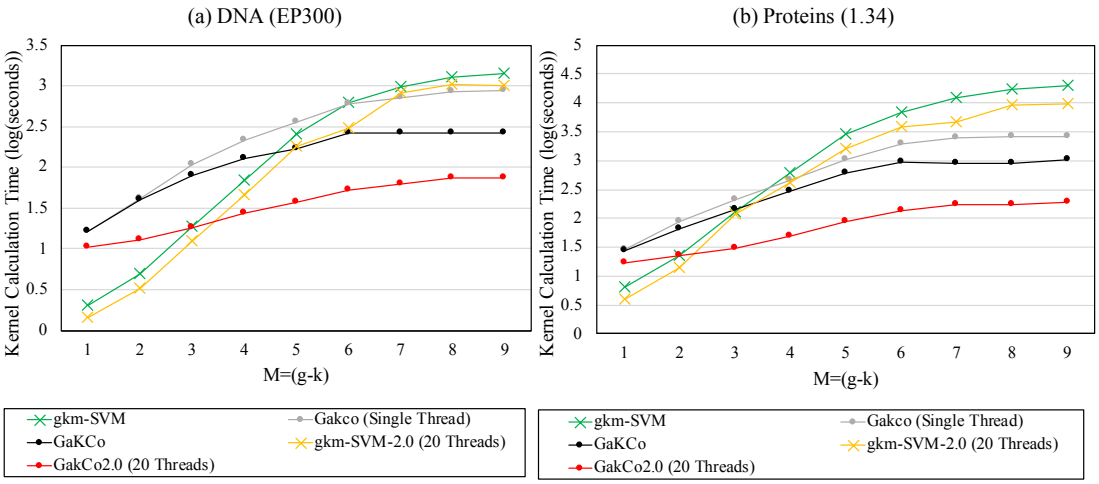

Figure 5.5: Comparison of kernel calculation times for multi-thread implementations

Kernel calculation profiles of original GaKCo versus GaKCo2.0 for varying values of mismatches $M = (g - k) = \{1, \ldots, 9\}$ and both (a) DNA dataset (EP300) and (b) Protein dataset (1.34). We also compare both the GaKCo implementations with gkm-SVM and its multi-thread version (gkm-SVM-2.0) and show that overall GaKCo2.0 performs the fastest for larger values of mismatches (i.e $M > 3$).

GaKCo scales better than gkm-SVM for increasing number of sequences (N):

We now compare the kernel calculation times of GaKCo versus gkm-SVM for increasing number of sequences (N). In Figure 5.6(c), we plot the kernel calculation times of GaKCo and gkm-SVM for best performing parameters (g, k) for three binary classification datasets: EP300 (DNA), 1.34 (protein), and Sentiment (text). We select these three datasets as they provide the best AUC scores out of all 19 tasks (see Table 5.4). To show the effect of increasing $N = \{100, 250, 500, 750\}$ on kernel calculation times, we fix the length of the sequences for all three datasets to $l = 100$. As expected, the time grows for both the algorithms with the increase in the number of sequences. However, this growth in time is more drastic for gkm-SVM than for GaKCo across all three datasets. Therefore, GaKCo is ideal for adaptive training since its kernel calculation time increases more gradually than gkm-SVM as new sequences are added.

GaKCo is both time and memory efficient:

Figure 5.7 (a), shows points for the kernel calculation time (X-axis) versus the memory usage (Y-axis) for both GaKCo and gkm-SVM for all 19 classification tasks. We observe that most of these points representing GaKCo lie in the **lower-left quadrant** indicating that it is both time and memory efficient. For 17/19

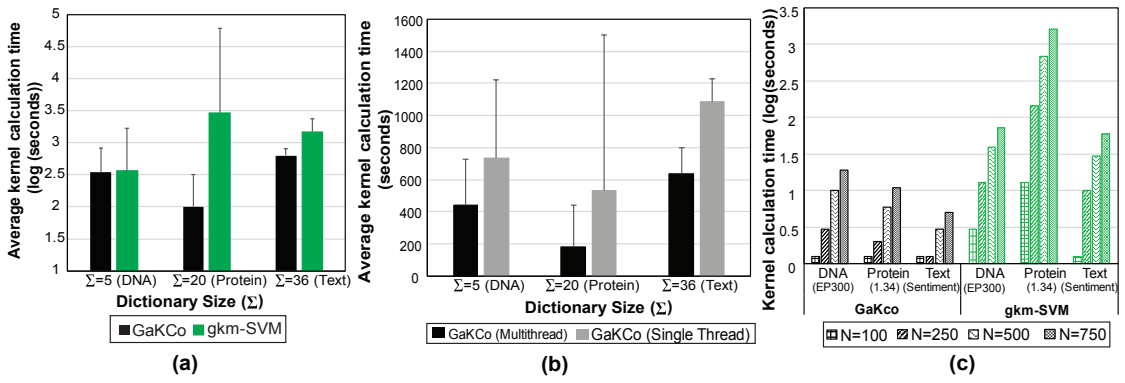

Figure 5.6: Further kernel calculation time analyses

(a) Average kernel calculation times (lower is better) for the best performing (g, k) parameters for DNA ($\Sigma = 5$), protein ($\Sigma = 20$), and text ($\Sigma = 36$) datasets. gkm-SVM takes similar time as GaKCo to calculate the kernel for DNA dataset but slows down considerably for protein and text datasets due to increase in dictionary size. Since GaKCo is independent of the dictionary size, it is consistently faster for all three datasets. (b) Average kernel calculation times (lower is better) across DNA (5), protein (12) and text (2) datasets. Multi-thread GaKCo implementation improves the kernel calculation speed of the single-thread GaKCo by a factor of 2. (c) Kernel calculation times (lower is better) of GaKCo and gkm-SVM for best performing parameters (g, k) for: EP300 (DNA), 1.34 (protein), and Sentiment (text) datasets. Length of the sequences for all three datasets is fixed to $l = 100$ and number of sequences are varied for $N = \{100, 250, 500, 750\}$. With increasing number of sequences, the increase in kernel calculation time is more drastic for gkm-SVM than for GaKCo across all three datasets.

tasks, its memory usage is lesser or comparable to gkm-SVM with faster kernel calculation time. Therefore, GaKCo's time improvement over the baseline is achieved with almost no added memory cost.

5.4.2 Empirical Performance of GaKCo versus NN

Figure 5.1 (b) demonstrated that GaKCo achieves same empirical performance as gkm-SVM (AUC scores or F1-score). This is because GaKCo's gapped k-mer formulation is same as gkm-SVM but with improved (faster) implementation. In this section, we compare GaKCo's empirical performance with state-of-the-art CNN model [8]. Figure 5.7 (b) shows the differences in AUC Scores (or micro-averaged F1-score for Web-KB) of GaKCo and CNN [8]. For 16/19 tasks, GaKCo outperforms the CNN model with an average of $\sim 20\%$ accuracy. This result can be explained by the fact that CNNs trained with a small number of samples (1000-10,000 sequences) often exhibit unstable behavior in performance.

For three datasets - SIN3A (DNA), 1.1 (protein), and Web-KB (text), we observe that the empirical performance of GaKCo and CNN is similar. Therefore, we further explore these datasets in Figure 5.7(c). Here, we plot the AUC scores or micro-averaged F1 scores (Web-KB) for varying number of training sample ($N = \{100, 250, 500$ and $750\}$ sequences). We randomly select these samples from the training set and use the original test set of the respective datasets. The results are averaged over three runs of the experiment.

Table 5.4: Summary of GaKCo, gkm-SVM and CNN-AUC scores for all datasets.

For Web-KB we report the micro-averaged F1-Score since it is a multi classification task with four classes: student, faculty, project and course.

Prediction Task	**Sample properties**			**Best Parameters**			**AUC**		
Datasets	N	Σ	Max(l)	g	k	c	GaKCO-AUC	gkm-SVM-AUC	NN-AUC
1.1	3574	20	905	7	5	0.01	0.7453	0.7448	0.7484
1.34	3312			10	1	0.1	0.9903	0.9903	0.9858
2.19	2560			7	1	100	0.8951	0.8951	0.822
2.31	3500			10	7	10	0.9484	0.9497	0.5317
2.1	7062			10	3	10	0.979	0.9895	0.7970
2.34	2738			7	6	0.01	0.8664	0.8660	0.7477
2.41	2646			10	6	0.01	0.7925	0.7925	0.6484
2.8	2480			10	1	10	0.6367	0.6367	0.6801
3.19	3341			8	1	0.1	0.9326	0.9326	0.7050
3.25	3637			10	8	1	0.7967	0.7962	0.5848
3.33	2918			10	5	0.01	0.9018	0.9018	0.8843
3.50	2549			10	7	0.01	0.7768	0.7772	0.8265
CTCF	4000	5	100	10	5	1	0.902	0.902	0.7834
EP300				10	5	1	0.942	0.942	0.6138
JUND				10	7	1	0.91	0.91	0.8317
RAD21				10	5	1	0.901	0.901	0.7937
SIN3A				10	7	1	0.834	0.834	0.8309
Sentiment	9217	36	260	8	4	1	0.8154	0.81	0.5303
WebKB (F1-score)	4163	36	14218	8	5	1	0.9153	0.9116	0.9147

We aim to find the threshold (number of training samples) for which CNN gives a lower performance to GaKCo for these three datasets. Figure 5.7(c) presents the averaged AUC scores or micro-averaged F1 score (Web-KB). We see that the threshold for which CNN gives a lower performance to GaKCo is 750 sequences in the training set. We also observe that the variance in performance is high for NN (represented by error bars) across the three runs.

In summary, the advantages of this work are:

- **Fast:** GaKCo is a novel combination of two efficient concepts: (1) reduced gapped k-mer feature space and (2) associative array based counting method, making it faster than the state-of-the-art gapped k-mer string kernel, while achieving same accuracy. (Figure 5.1).
- GaKCo can **scale up** to larger values of m and Σ. (Figure 5.4 and Figure 5.6(a))
- **Parallelizable:** GaKCo algorithm naturally leads to a parallelizable implementation (Figure 5.4 and Figure 5.6 (b))

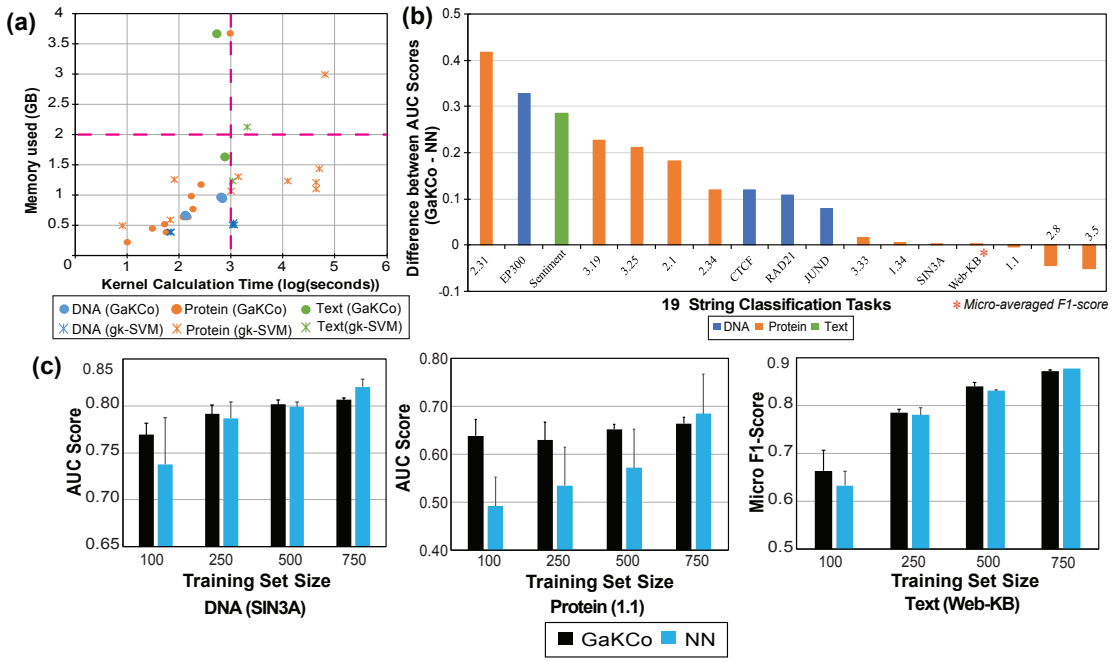

Figure 5.7: Time versus Memory comparison and GaKCo versus DNN.

(a) Kernel calculation time (X-axis) and the memory usage (Y-axis) (lower is better) for both GaKCo and gkm-SVM for all 19 classification tasks. For 17/19 tasks, GaKCo's memory usage is lesser or comparable to gkm-SVM with lower kernel calculation time. Therefore, it is both time and memory efficient. (b) Differences in AUC Scores (or micro-averaged F1-score for Web-KB) between GaKCo and state-of-the-art CNN model [8]. For 16/19 tasks, GaKCo outperforms CNN with an average of $\sim 20\%$ accuracy. (c)Averaged AUC scores, across 3 runs, for SIN3A (DNA) and 1.1 (protein), and micro-averaged F1 scores for Web-KB (text) while varying number of sequences ($N = \{100, 250, 500, \text{and } 750\}$). For a threshold value of 750 sequences in the training set, CNN achieves lower empirical performance to GaKCo.

- We have provided a detailed **theoretical analysis** comparing the asymptotic time complexity of GaKCo with gkm-SVM. This analysis, to the best of the authors' knowledge, has not been reported before (Section 5.2.3).

Chapter 6

Conclusion and Future Work

6.1 Intellectual Merit

The proposed DeepChrome and AttentiveChrome models are novel applications of DNNs for gene expression prediction. To our knowledge, we are the first to introduce the use of attention mechanism for this particular task. AttentiveChrome is a unified, end-to-end architecture that not only provides accurate predictions but also allows interpretation by enabling us to visualize the features that are necessary for a particular prediction.

GaKCo implementation combines two highly efficient approaches: (1) gapped k-mer kernel, which considerably reduces the feature space and (2) counting based algorithm, that is independent of the dictionary size and easily parallelizable. Our results indicate that this novel combination significantly reduces the kernel calculation time of GaKCo. It is an improvement over the state-of-the-art SK-SVM techniques and a complementary method to neural networks when the sample size is small (< 5000 sequences).

6.2 Future Work

6.2.1 Extension of DNNs for Gene Expression Prediction

Regression Task We acknowledge that formulating gene expression prediction as a binary classification simplifies the task. Thus, a natural extension to our work is converting the classification setup to regression setup. This can be achieved by replacing the *softmax* function and NLL loss by Mean Squared Error (MSE) loss. To this end, We scaled the gene expression values using *log* function. We implemented the proposed changes for DeepChrome to perform regression and computed the Pearson Correlation Coefficients of predictions with actual expression values. Next, we compared these with the Support Vector Regression (SVR) baseline using the best-bin strategy and found that DeepChrome for regression outperforms the baseline for 47/56 cell types (Figure 6.1). Future work in this direction would involve refining the model further as well as implementing regression for AttentiveChrome to generate more informative attention maps.

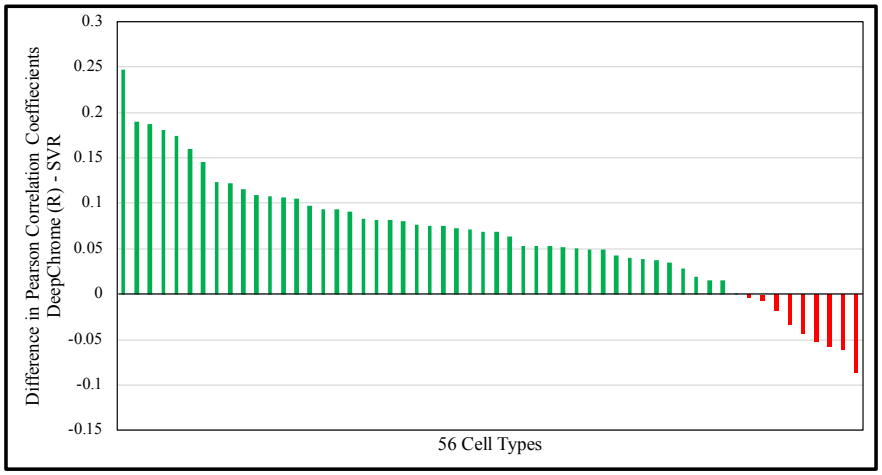

Figure 6.1: Comparison of DeepChrome (Regression) versus Support Vector Regression (SVR) baseline

We calculated the Pearson Correlation Coefficients of predictions from DeepChrome and the SVM baseline with actual expression values for the gene expression prediction task. DeepChrome for regression (DeepChrome (R)) outperforms the baseline for 47/56 cell types.

Combining new information Gene regulation involves complex cellular machinery. There is a multitude of factors apart from histone modifications that can affect the gene regulations. For example, Transcription factors (TFs) are regulatory proteins that bind to specific locations (or sites) on a genomic sequence to regulate the gene. Similarly, 'open' and 'close' regions of the DNA control the binding of proteins and in turn control gene expression. New sequencing methods have allowed biologists to collect all this information in the

form of sequential data and the deep learning community has developed different models complex enough to extract meaningful representations from it. Therefore, another relevant extension of AttentiveChrome is combining all the different datasets (TF-binding, HMs, Open/Close region information, DNA sequences, etc.) as inputs and improving modeling and interpretation to predict and understand the gene regulation more comprehensively.

6.2.2 Scalability of SK-SVM Methods

Despite the state-of-the-art performance, string kernel methods (including GaKCo) suffer from scalability issues when dealing with a large number of training samples (N). The string kernel SVM solver can become prohibitively expensive in execution time (kernel computations proportional to N^2) as well as memory (N^2 storage space for pre-computed kernel). This scalability issue of string kernels can prove to be a bottleneck for advancing the frontiers of large-scale sequence classifications.

As a solution to this problem, Lee et al. [100] implemented the gapped k-mer kernel functions within the LIBSVM framework [98]. LIBSVM uses decomposition methods for SVM training. The decomposition method [101] iteratively finds and solves a small subset of SVM problems that only requires a partial kernel matrix. In case of LIBSVM, the solver requires only two columns of the kernel matrix at a time. Therefore, Lee et al. [100] replace the LIBSVM kernel routines with the gkm-SVM kernel functions, eliminating the requirement to pre-compute the entire $N \times N$ matrix. This allows training on the larger number of samples as each column of the kernel is calculated online when the SVM is training. However, since the kernel calculation is gkm-SVM's function, this implementation still suffers from Σ^m computation term and has not be extended for cases other than DNA sequences ($\Sigma = 4$).

A similar decomposition strategy using GaKCo's kernel function would be useful improving its scalability. To this end, we have merged the pre-computed GaKCo kernel calculation and SVM training into one single program. This makes it easier for researchers to do the whole training and testing pipeline, starting with labeled data and easily producing predictions for unlabeled datasets. However, the adoption of the SVM decomposition with GaKCo kernel calculation remains an open problem, and its solution will help in increasing the popularity of string kernel-based methods for large-scale sequence classification tasks.

6.3 Broader Impact

This research is a successful amalgamation of important aspects of both biology and machine learning. The primary aim of this work is to collaborate with biologists in hypothesis development and planning experiments. In the process of solving challenges associated with sequential datasets in biology, we further improve the state-of-the-art machine learning methods. While Attentive-Chrome can visualize relevant features for 30,000 genes across 56 cell-types, GaKCo can enhance the speed of string kernel calculation by a factor of 100. With the help of interpretable and fast tools like AttentiveChrome and GaKCo, we hope to provide a better understanding of underlying mechanisms in biology.

Appendix

A:1 Selecting input HM features for DNNs

Not all HMs carry the same information, and it is important to include different HMs for gene expression prediction. While $H3K4me3$ may be essential to predict gene=ON, $H3K4me1$ may play a role to make that prediction. Contrarily, for OFF genes, HMs like $H3K27me3$ may play a significant role. To demonstrate this, we used only one HM at a time and performed the classification using AttentiveChrome. The accuracy decreases when just one HM is used. Table A:1 shows AUC scores in GM12878 when all HMs are used as input signals and when we use them one at a time. We observe that the performance drops drastically, indicating that it is vital to include different HMs for gene expression prediction.

Table A:1: AUC scores in GM12878 when each HM is used as input signal one at a time.

The AUC score reduces drastically, indicating that it is vital to include different HMs for gene expression prediction

HMs used as input	**AUC Score**
All 5 HMs	0.9085
$H3K4me3$	0.8893
$H3K4me1$	0.8516
$H3K36me3$	0.8506
$H3K27me3$	0.7698
$H3K9me3$	0.6465

We also performed feature selection of HMs, such that we removed one HM at a time for GM12878 cell type and observed the change accuracy performance of AttentiveChrome model (Table A:2). We find that $H3K4me3$ and $H3K27me3$ are the most important signals for gene expression prediction as their removal causes the highest dip in AUC scores. Comparing these results with Table A:1 also indicates that while

$H3K27me3$ alone might not have the best predictive power, when combined with other HM signals, it is vital for accurate predictions.

Table A:2: Variations in AUC scores in GM12878 when one HM is removed from the input one at a time.

$H3K4me3$ and $H3K27me3$ are the most important signals for gene expression prediction as their removal causes the highest dip in AUC scores.

Removed HMs	**AUC Score**
None	0.9085
$H3K4me3$	**0.8960**
$H3K4me1$	0.9048
$H3K36me3$	0.9069
$H3K27me3$	**0.8942**
$H3K9me3$	0.8961

A previous study [66] combined 1 HM signal at a time using linear regression and established that not all modifications are equally important due to a certain level of *redundancy* in the information that HMs provide. They showed that a combination of only two to three specific modifications is sufficient for making accurate predictions. However, they also showed that adding more HMs helps in predictions. Our observations are consistent with findings of this study.

A:2 Formal proof regarding Hamming Distance Property

Let hamming distance between strings x and y be $d(x, y)$. Assuming both x and y are composed of n characters, then hamming distance is formally defined as [54]:

$$d(x, y) = \Sigma_{i=0}^{n} neq(x_i, y_i) \tag{A:1}$$

where, if a and b are two characters,

$$neq(a, b) = \begin{cases} 0, & \text{if } a = b \\ 1, & \text{otherwise} \end{cases} \tag{A:2}$$

Property: Given, there are two strings x and y (composed of n characters each) and characters from p positions are removed to obtain strings x' and y' with $(n-p)$ characters. If the hamming distance between x' and y', $d(x', y') = 0$ then the hamming distance between original x and y, $d(x, y) \leq p$.

Proof by example:

Let $p = 2$. We first re-write Eq. A:1 as:

$$d(x, y) = \Sigma_{i=0}^{n-2} neq(x_i, y_i) + neq(x_{n-1}, y_{n-1}) + neq(x_n, y_n) \tag{A:3}$$

That is, we split the summation of $neq(.)$ function as summation of $neq(.)$ for $(n-2)$ characters plus the sum of $neq(.)$ for the $(n-1)^{th}$ and last n^{th} character for x and y.

The term $\Sigma_{i=0}^{n-2} neq(x_i, y_i)$ represents the hamming distance $d(x', y')$ for $p = 2$ positions removed. Therefore:

$$d(x, y) = d(x', y') + neq(x_{n-1}, y_{n-1}) + neq(x_n, y_n) \tag{A:4}$$

Now if $d(x', y') = 0$ then

$$d(x, y) = neq(x_{n-1}, y_{n-1}) + neq(x_n, y_n) \tag{A:5}$$

Based on Eq. A:2, $d(x, y) = \{(0+0), (0+1), (1+0), (1+1)\}$ as these are all the possible values of $neq(.)$ function.

Therefore, $d(x, y) \leq 2$ if $d(x', y') = 0$ where x' and y' are x and y (respectively) with characters removed from $p = 2$ positions.

Bibliography

[1] Anshul Kundaje, Wouter Meuleman, Jason Ernst, Misha Bilenky, Angela Yen, Alireza Heravi-Moussavi, Pouya Kheradpour, Zhizhuo Zhang, Jianrong Wang, Michael J Ziller, et al. Integrative analysis of 111 reference human epigenomes. *Nature*, 518(7539):317–330, 2015.

[2] P Borah. Biological databases with emphasis on biodiversity and conservation. 2011.

[3] Vivien Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

[4] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[5] Xianjun Dong and Zhiping Weng. The correlation between histone modifications and gene expression. *Epigenomics*, 5(2):113–116, 2013.

[6] Ritambhara Singh, Jack Lanchantin, Gabriel Robins, and Yanjun Qi. Deepchrome: deep-learning for predicting gene expression from histone modifications. *Bioinformatics*, 32(17):i639–i648, 2016.

[7] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of DNA- and RNA- binding proteins by deep learning. *Nature Biotechnology*, 2015.

[8] Jack Lanchantin, Ritambhara Singh, Beilun Wang, and Yanjun Qi. Deep motif dashboard: Visualizing and understanding genomic sequences using deep neural networks. *arXiv preprint arXiv:1608.03644*, 2016.

[9] Yanjun Qi, Merja Oja, Jason Weston, and William Stafford Noble. A unified multitask architecture for predicting local protein properties. *PloS One*, 7(3):e32235, 2012.

[10] Jian Zhou and Olga G Troyanskaya. Deep supervised and convolutional generative stochastic network for protein secondary structure prediction. *arXiv preprint arXiv:1403.1347*, 2014.

[11] Christina Leslie and Rui Kuang. Fast string kernels using inexact matching for protein sequences. *The Journal of Machine Learning Research*, 5:1435–1455, 2004.

[12] Pavel P. Kuksa, Pai-Hsi Huang, and Vladimir Pavlovic. Scalable algorithms for string kernels with inexact matching. In *NIPS'08*, pages 881–888, 2008.

[13] Aaron Arvey, Phaedra Agius, William Stafford Noble, and Christina Leslie. Sequence and chromatin determinants of cell-type–specific transcription factor binding. *Genome research*, 22(9):1723–1734, 2012.

[14] Manu Setty and Christina S Leslie. Seqgl identifies context-dependent binding signals in genome-wide regulatory element maps. *PLoS Comput Biol*, 11(5):e1004271, 2015.

[15] Ritambhara Singh, Jack Lanchantin, Gabriel Robins, and Yanjun Qi. Transfer string kernel for cross-context dna-protein binding prediction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[16] Ritambhara Singh and Yanjun Qi. Character based string kernels for bio-entity relation detection. *ACL 2016*, page 66, 2016.

[17] Mahmoud Ghandi, Dongwon Lee, Morteza Mohammad-Noori, and Michael A Beer. Enhanced regulatory sequence prediction using gapped k-mer features. *PLoS Comput Biol*, 10(7):e1003711, 2014.

[18] Tommi Jaakkola, Mark Diekhans, and David Haussler. A discriminative framework for detecting remote protein homologies. *Journal of computational biology*, 7(1-2):95–114, 2000.

[19] James M Ostell, Sarah J Wheelan, and Jonathan A Kans. The ncbi data model. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, 43:19, 2004.

[20] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[22] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[23] Zeming Lin, Jack Lanchantin, and Yanjun Qi. MUST-CNN: A multilayer shift-and-stitch deep convolutional architecture for sequence-based protein structure prediction. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016.

[24] Michael KK Leung, Hui Yuan Xiong, Leo J Lee, and Brendan J Frey. Deep learning of the tissue-regulated splicing code. *Bioinformatics*, 30(12):i121–i129, 2014.

[25] Jack Lanchantin, Ritambhara Singh, Zeming Lin, and Yanjun Qi. Deep motif: Visualizing genomic sequence classifications. In *Workshop Track 4th International Conference on Learning Representations (ICLR 2016)*, 2016.

[26] Jian Zhou and Olga G Troyanskaya. Predicting effects of noncoding variants with deep learning-based sequence model. *Nature Methods*, 12(10):931–934, 2015.

[27] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. 2017.

[28] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*, pages 818–833. Springer, 2014.

[29] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. Visualizing deep neural network decisions: Prediction difference analysis. *arXiv preprint arXiv:1702.04595*, 2017.

[30] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[31] David Baehrens, Timon Schroeter, Stefan Harmeling, Motoaki Kawanabe, Katja Hansen, and Klaus-Robert MÃžller. How to explain individual classification decisions. volume 11, pages 1803–1831, 2010.

[32] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[33] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. volume 10, page e0130140, 2015.

[34] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Gradients of counterfactuals. *arXiv preprint arXiv:1611.02639*, 2016.

[35] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. 2015.

[36] Jiwei Li, Xinlei Chen, Eduard Hovy, and Dan Jurafsky. Visualizing and understanding neural models in nlp. 2015.

[37] Maurizio Corbetta and Gordon L Shulman. Control of goal-directed and stimulus-driven attention in the brain. *Nature reviews neuroscience*, 3(3):201–215, 2002.

[38] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[39] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention.

[40] Volodymyr Mnih, Nicolas Heess, Alex Graves, and others. Recurrent models of visual attention. In *Advances in neural information processing systems*, pages 2204–2212.

[41] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *ICML*, volume 14, pages 77–81, 2015.

[42] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. Hierarchical attention networks for document classification. 2016.

[43] Li Yao, Atousa Torabi, Kyunghyun Cho, Nicolas Ballas, Christopher Pal, Hugo Larochelle, and Aaron Courville. Describing videos by exploiting temporal structure. In *Computer Vision (ICCV), 2015 IEEE International Conference on*. IEEE, 2015.

[44] Huijuan Xu and Kate Saenko. Ask, attend and answer: Exploring question-guided spatial attention for visual question answering. In *ECCV*, 2016.

[45] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015.

[46] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1412–1421, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[47] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 577–585. Curran Associates, Inc., 2015.

[48] Vladimir N. Vapnik. *Statistical Learning Theory*. Wiley-Interscience, September 1998.

[49] Christina S. Leslie, Eleazar Eskin, and William Stafford Noble. The spectrum kernel: A string kernel for svm protein classification. In *Pacific Symposium on Biocomputing*, pages 566–575, 2002.

[50] G Rätsch and S Sonnenburg. Accurate splice site prediction for caenorhabditis elegans, 277–298, 2004.

[51] SVN Vishwanathan, Alexander Johannes Smola, et al. Fast kernels for string and tree matching. *Kernel methods in computational biology*, pages 113–130, 2004.

[52] Rui Kuang, Eugene Ie, Ke Wang, Kai Wang, Mahira Siddiqi, Yoav Freund, and Christina Leslie. Profile-based string kernels for remote homology detection and motif extraction. *Journal of bioinformatics and computational biology*, 3(03):527–550, 2005.

[53] Olivier Chapelle, Jason Weston, and Bernhard Schölkopf. Cluster kernels for semi-supervised learning. In *Advances in neural information processing systems*, pages 585–592, 2002.

[54] Amihood Amir, Moshe Lewenstein, and Ely Porat. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257–275, 2004.

[55] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.

[56] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, Mu-Tian Chen, and Joel Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical computer science*, 40:31–55, 1985.

[57] William I Chang and Eugene L Lawler. Sublinear expected time approximate string matching and biological. 1991.

[58] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[59] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[60] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[61] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[62] Sanguthevar Rajasekaran, Sudha Balla, and C-H Huang. Exact algorithms for planted motif problems. *Journal of Computational Biology*, 12(8):1117–1128, 2005.

[63] Andrew J Bannister and Tony Kouzarides. Regulation of chromatin by histone modifications. *Cell research*, 21(3):381–395, 2011.

[64] Pek S Lim, Kristine Hardy, Karen L Bunting, Lina Ma, Kaiman Peng, Xinxin Chen, and Mary F Shannon. Defining the chromatin signature of inducible genes in T cells. *Genome Biology*, 10(10):R107, 2009.

[65] Carolyn E Cain, Ran Blekhman, John C Marioni, and Yoav Gilad. Gene expression differences among primates are associated with changes in a histone epigenetic modification. *Genetics*, 187(4):1225–1234, 2011.

[66] Rosa Karlić, Ho-Ryun Chung, Julia Lasserre, Kristian Vlahoviček, and Martin Vingron. Histone modification levels are predictive for gene expression. *Proceedings of the National Academy of Sciences*, 107(7):2926–2931, 2010.

[67] Zhibin Wang, Chongzhi Zang, Jeffrey A Rosenfeld, Dustin E Schones, Artem Barski, Suresh Cuddapah, Kairong Cui, Tae-Young Roh, Weiqun Peng, Michael Q Zhang, et al. Combinatorial patterns of histone acetylations and methylations in the human genome. *Nature Genetics*, 40(7):897–903, 2008.

[68] Ivan G Costa, Helge G Roider, Thais G Rego, and Francisco AT Carvalho. Predicting gene expression in T cell differentiation from histone modifications and transcription factor binding affinities by linear mixture models. *BMC Bioinformatics*, 12(1):1, 2011.

[69] Chao Cheng, Koon-Kiu Yan, Kevin Y Yip, Joel Rozowsky, Roger Alexander, Chong Shou, Mark Gerstein, et al. A statistical framework for modeling gene expression using chromatin features and application to modENCODE datasets. *Genome Biology*, 12(2):R15, 2011.

[70] Susan E Celniker, Laura AL Dillon, Mark B Gerstein, Kristin C Gunsalus, Steven Henikoff, Gary H Karpen, Manolis Kellis, Eric C Lai, Jason D Lieb, David M MacAlpine, et al. Unlocking the secrets of the genome. *Nature*, 459(7249):927–930, 2009.

[71] Xianjun Dong, Melissa C Greven, Anshul Kundaje, Sarah Djebali, James B Brown, Chao Cheng, Thomas R Gingeras, Mark Gerstein, Roderic Guigó, Ewan Birney, et al. Modeling gene expression using chromatin features in various cellular contexts. *Genome Biology*, 13(9):R53, 2012.

[72] ENCODE Project Consortium et al. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[73] Bich Hai Ho, Rania Mohammed Kotb Hassen, and Ngoc Tu Le. Combinatorial roles of DNA methylation and histone modifications on gene expression. In *Some Current Advanced Researches on Information and Computer Science in Vietnam*, pages 123–135. Springer, 2015.

[74] Jason Ernst and Manolis Kellis. Large-scale imputation of epigenomic datasets for systematic annotation of diverse human tissues. *Nature Biotechnology*, 33(4):364–376, 2015.

[75] Vibhor Kumar, Masafumi Muratani, Nirmala Arul Rayan, Petra Kraus, Thomas Lufkin, Huck Hui Ng, and Shyam Prabhakar. Uniform, optimal signal processing of mapped deep-sequencing data. *Nature Biotechnology*, 31(7):615–622, 2013.

[76] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[77] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[78] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[79] Léon Bottou. Stochastic learning. In *Advanced Lectures on Machine Learning*, pages 146–168. Springer, 2004.

[80] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[81] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[82] Efthymia Papalexi and Rahul Satija. Single-cell rna sequencing to explore immune cell heterogeneity. *Nature Reviews Immunology*, 18(1):35, 2018.

[83] Joanna Boros, Nausica Arnoult, Vincent Stroobant, Jean-François Collet, and Anabelle Decottignies. Polycomb repressive complex 2 and H3K27me3 cooperate with H3K9 methylation to maintain heterochromatin protein 1α at chromatin. *Molecular and Cellular Biology*, 34(19):3662–3674, 2014.

[84] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[85] Shane McManus, Anja Ebert, Giorgia Salvagiotto, Jasna Medvedovic, Qiong Sun, Ido Tamir, Markus Jaritz, Hiromi Tagoh, and Meinrad Busslinger. The transcription factor pax5 regulates its target genes by recruiting chromatin-modifying proteins in committed b cells. *The EMBO journal*, 30(12):2388–2404, 2011.

[86] Yi Zhang and Danny Reinberg. Transcription regulation by histone methylation: interplay between different covalent modifications of the core histone tails. *Genes & development*, 15(18):2343–2360, 2001.

[87] Tianyi Zhang, Sarah Cooper, and Neil Brockdorff. The interplay of histone modifications–writers that read. *EMBO reports*, 16(11):1467–1481, 2015.

[88] Peter J Park. Chip–seq: advantages and challenges of a maturing technology. *Nature Reviews Genetics*, 10(10):669–680, 2009.

[89] Pierre Baldi and Søren Brunak. *Bioinformatics: the machine learning approach*. MIT press, 2001.

[90] Mahmoud Ghandi, Morteza Mohammad-Noori, and Michael A Beer. Robust k-mer frequency estimation using gapped k-mers. *Journal of mathematical biology*, 69(2):469–500, 2014.

[91] Ana Cardoso-Cachopo. Improving Methods for Single-label Text, Categorization. PdD Thesis, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, 2007.

[92] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[93] Miguel Ballesteros, Chris Dyer, and Noah A Smith. Improved transition-based parsing by modeling characters instead of words with lstms. *arXiv preprint arXiv:1508.00657*, 2015.

[94] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2(Feb):419–444, 2002.

[95] Pavel P. Kuksa, Pai-Hsi Huang, and Vladimir Pavlovic. Efficient use of unlabeled data for protein sequence classification: a comparative study. *BMC Bioinformatics*, 10(S-4), 2009.

[96] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, Christopher Potts, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer, 2013.

[97] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[98] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[99] Mahmoud Ghandi, Morteza Mohammad-Noori, Narges Ghareghani, Dongwon Lee, Levi Garraway, and Michael A Beer. gkmsvm: an r package for gapped-kmer svm. *Bioinformatics*, 32(14):2205–2207, 2016.

[100] Dongwon Lee. Ls-gkm: a new gkm-svm for large-scale datasets. *Bioinformatics*, 32(14):2196–2198, 2016.

[101] Rong-En Fan, Pai-Hsuen Chen, and Chih-Jen Lin. Working set selection using second order information for training support vector machines. *Journal of machine learning research*, 6(Dec):1889–1918, 2005.

[102] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[103] Pedro HO Pinheiro and Ronan Collobert. Recurrent convolutional neural networks for scene parsing. *arXiv preprint arXiv:1306.2795*, 2013.

[104] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[105] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[106] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, pages 160–167. ACM, 2008.

[107] E Eskin and E Agichtein. Combining text mining and sequence analysis to. In *Pacific Symposium on Biocomputing 2004: Hawaii, USA, 6-10 January 2004*, page 288. World Scientific, 2003.

[108] Daniel Blankenberg, James Taylor, Ian Schenck, Jianbin He, Yi Zhang, Matthew Ghent, Narayanan Veeraraghavan, Istvan Albert, Webb Miller, Kateryna D Makova, et al. A framework for collaborative analysis of encode data: making large-scale analyses biologist-friendly. *Genome research*, 17(6):960–964, 2007.

[109] Enis Afgan, Dannon Baker, Marius van den Beek, Daniel Blankenberg, Dave Bouvier, Martin Čech, John Chilton, Dave Clements, Nate Coraor, Carl Eberhard, et al. The galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. *Nucleic acids research*, page gkw343, 2016.

[110] Maxwell W Libbrecht and William Stafford Noble. Machine learning applications in genetics and genomics. *Nature Reviews Genetics*, 16(6):321–332, 2015.

[111] Stephen G Landt, Georgi K Marinov, Anshul Kundaje, Pouya Kheradpour, Florencia Pauli, Serafim Batzoglou, Bradley E Bernstein, Peter Bickel, James B Brown, Philip Cayting, et al. Chip-seq guidelines and practices of the encode and modencode consortia. *Genome research*, 22(9):1813–1831, 2012.

[112] Domonkos Tikk, Philippe Thomas, Peter Palaga, Jörg Hakenberg, and Ulf Leser. A comprehensive benchmark of kernel methods to extract protein–protein interactions from literature. *PLoS Comput Biol*, 6(7):e1000837, 2010.

[113] Martin Krallinger, Florian Leitner, Carlos Rodriguez-Penagos, Alfonso Valencia, et al. Overview of the protein-protein interaction annotation extraction task of BioCreative II. *Genome biology*, 9(Suppl 2):S4, 2008.

[114] Karsten M Borgwardt. Kernel methods in bioinformatics. In *Handbook of Statistical Bioinformatics*, pages 317–334. Springer, 2011.

[115] Tommi Jaakkola, Mark Diekhans, and David Haussler. Using the Fisher kernel method to detect remote protein homologies. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 149–158. AAAI Press, 1999.

[116] Tommi Jaakkola, Mark Diekhans, and David Haussler. A discriminative framework for detecting remote protein homologies. In *Journal of Computational Biology*, volume 7, pages 95–114, 2000.

[117] T. Joachims. Making large-scale support vector machine learning practical. In A. Smola B. Schölkopf, C. Burges, editor, *Advances in Kernel Methods: Support Vector Machines*. MIT Press, Cambridge, MA, 1998.

[118] Christina S. Leslie, Eleazar Eskin, Jason Weston, and William Stafford Noble. Mismatch string kernels for svm protein classification. In *NIPS*, pages 1417–1424, 2002.

[119] Jason Weston, Christina Leslie, Eugene Ie, Dengyong Zhou, Andre Elisseeff, and William Stafford Noble. Semi-supervised protein classification using cluster kernels. *Bioinformatics*, 21(15):3241–3247, 2005.

[120] M. Gribskov and N. Robinson. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching, 1996.

[121] Steven Henikoff and Jorja G. Henikoff. Position-based sequence weights. *J Mol Biol.*, 243(4):574–8, 11 1994.

[122] Koji Tsuda, Taishin Kin, and Kiyoshi Asai. Marginalized kernels for biological sequences. *Bioinformatics*, 18(suppl_1):S268–275, 2002.

[123] Benjamin Schuster-Bockler, Jorg Schultz, and Sven Rahmann. Hmm logos for visualization of protein families. *BMC Bioinformatics*, 5(1):7, 2004.

[124] Li Liao and William Stafford Noble. Combining pairwise sequence similarity and support vector machines for remote protein homology detection. In *RECOMB*, pages 225–232, 2002.

[125] Sören Sonnenburg, Gunnar Rätsch, and Bernhard Schölkopf. Large scale genomic sequence svm classifiers. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 848–855, New York, NY, USA, 2005.

[126] Bernhard Schölkopf, Koji Tsuda, and Jean-Philippe Vert. *Kernel methods in computational biology*. MIT press, 2004.

[127] Rui Kuang, Eugene Ie, Ke Wang, Kai Wang, Mahira Siddiqi, Yoav Freund, and Christina Leslie. Profile-based string kernels for remote homology detection and motif extraction. *J Bioinform Comput Biol*, 3(3):527–550, June 2005.

[128] Christina Leslie and Rui Kuang. Fast string kernels using inexact matching for protein sequences. *J. Mach. Learn. Res.*, 5:1435–1455, 2004.

[129] Pavel Kuksa, Pai-Hsi Huang, and Vladimir Pavlovic. Fast and accurate multi-class protein fold recognition with spatial sample kernels. In *Computational Systems Bioinformatics: Proceedings of the CSB2008 Conference*, pages 133–143, 2008. Acceptance rate: 30/135 (22

[130] Timothy L Bailey, Mikael Boden, Fabian A Buske, Martin Frith, Charles E Grant, Luca Clementi, Jingyuan Ren, Wilfred W Li, and William S Noble. Meme suite: tools for motif discovery and searching. *Nucleic acids research*, 37(suppl 2):W202–W208, 2009.

[131] Ivan V Kulakovskiy, VA Boeva, Alexander V Favorov, and VJ Makeev. Deep and wide digging for binding motifs in chip-seq data. *Bioinformatics*, 26(20):2622–2623, 2010.

[132] Theodoros Damoulas and Mark A. Girolami. Probabilistic multi-class multi-kernel learning: on protein fold recognition and remote homology detection. *Bioinformatics*, 24(10):1264–1270, 2008.

[133] Mohammad Tabrez Anwar Shamim, Mohammad Anwaruddin, and H.A. Nagarajaram. Support Vector Machine-based classification of protein folds using the structural properties of amino acid residues and amino acid residue pairs. *Bioinformatics*, 23(24):3320–3327, 2007.

[134] Hongzhu Qu and Xiangdong Fang. A brief review on the human encyclopedia of dna elements (encode) project. *Genomics, proteomics & bioinformatics*, 11(3):135–141, 2013.

[135] Kate R Rosenbloom, Cricket A Sloan, Venkat S Malladi, Timothy R Dreszer, Katrina Learned, Vanessa M Kirkup, Matthew C Wong, Morgan Maddren, Ruihua Fang, Steven G Heitner, et al. Encode data in the ucsc genome browser: year 5 update. *Nucleic acids research*, 41(D1):D56–D63, 2013.

[136] T. Joachims. Making large-scale svm learning practical. LS8-Report 24, Universität Dortmund, LS VIII-Report, 1998.

[137] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*, chapter 11, pages 169–184. MIT Press, Cambridge, MA, 1999.

[138] Daniel Quang and Xiaohui Xie. Danq: a hybrid convolutional and recurrent deep neural network for quantifying the function of dna sequences. *Nucleic acids research*, 44(11):e107–e107, 2016.

[139] Amartya Sanyal, Bryan R Lajoie, Gaurav Jain, and Job Dekker. The long-range interaction landscape of gene promoters. *Nature*, 489(7414):109–113, 2012.

[140] Jian Qun Ling and Andrew R Hoffman. Epigenetics of long-range chromatin interactions. *Pediatric research*, 61:11R–16R, 2007.

[141] Zhang Yan and Dianjing Guo. Comparative epigenetics analyses of acute and chronic leukemia. *Journal of Biosciences and Medicines*, 3(07):9, 2015.

[142] Inderpreet Sur and Jussi Taipale. The role of enhancers in cancer. *Nature Reviews Cancer*, 16(8):483–493, 2016.

[143] R David Hawkins, Gary C Hon, Chuhu Yang, Jessica E Antosiewicz-Bourget, Leonard K Lee, Que-Minh Ngo, Sarit Klugman, Keith A Ching, Lee E Edsall, Zhen Ye, et al. Dynamic chromatin states in human es cells reveal potential regulatory sequences and genes involved in pluripotency. *Cell research*, 21(10):1393–1409, 2011.

[144] Dennis Wang, Augusto Rendon, Willem Ouwehand, and Lorenz Wernisch. Transcription factor co-localization patterns affect human cell type-specific gene expression. *BMC genomics*, 13(1):263, 2012.

[145] Koichi Onodera, Tohru Fujiwara, Yasushi Onishi, Ari Itoh-Nakadai, Yoko Okitsu, Noriko Fukuhara, Kenichi Ishizawa, Ritsuko Shimizu, Masayuki Yamamoto, and Hideo Harigae. Gata2 regulates dendritic cell differentiation. *Blood*, 128(4):508–518, 2016.

[146] Xiangfan Liu, Huapeng Li, Mihir Rajurkar, Qi Li, Jennifer L Cotton, Jianhong Ou, Lihua J Zhu, Hira L Goel, Arthur M Mercurio, Joo-Seop Park, et al. Tead and ap1 coordinate transcription and motility. *Cell reports*, 14(5):1169–1180, 2016.

[147] Nadine Obier, Pierre Cauchy, Salam A Assi, Jane Gilmour, Michael Lie-A-Ling, Monika Lichtinger, Maarten Hoogenkamp, Laura Noailles, Peter N Cockerill, Georges Lacaud, et al. Cooperative binding of ap-1 and tead4 modulates the balance between vascular smooth muscle and hemogenic cell fate. *Development*, pages dev–139857, 2016.

[148] Aaron R Quinlan and Ira M Hall. Bedtools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2010.

[149] Daniel Quang and Xiaohui Xie. Danq: a hybrid convolutional and recurrent deep neural network for quantifying the function of dna sequences. page 032821. Cold Spring Harbor Labs Journals, 2015.

[150] Jack Lanchantin, Ritambhara Singh, Zeming Lin, and Yanjun Qi. Deep motif: Visualizing genomic sequence classifications. 2016.

[151] Dashboard definiton. `http://www.dictionary.com/browse/dashboard`. Accessed: 2016-07-20.

[152] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. 2015.

[153] Michael KK Leung, Andrew Delong, Babak Alipanahi, and Brendan J Frey. Machine learning in genomic medicine: A review of computational problems and data sets. IEEE, 2016.

[154] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[155] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.

[156] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. volume 5, pages 157–166. IEEE, 1994.

[157] David R Kelley, Jasper Snoek, and John L Rinn. Basset: Learning the regulatory code of the accessible genome with deep convolutional neural networks. Cold Spring Harbor Lab, 2016.

[158] Luisa M Zintgraf, Taco S Cohen, and Max Welling. A new method to visualize deep neural networks. 2016.

[159] Michael C O'Neill. Training back-propagation neural networks to define and detect dna-binding sites. volume 19, pages 313–318. Oxford Univ Press, 1991.

[160] Paul B Horton and Minoru Kanehisa. An assessment of neural network and statistical approaches for prediction of e. coli promoter sites. volume 20, pages 4331–4338. Oxford Univ Press, 1992.

[161] Philip Machanick and Timothy L Bailey. Meme-chip: motif analysis of large dna datasets. volume 27, pages 1696–1697. Oxford Univ Press, 2011.

[162] David Bisant and Jacob Maizel. Identification of ribosome binding sites in escherichia coli using neural network models. volume 23, pages 1632–1639. Oxford Univ Press, 1995.

[163] Aravindh Mahendran and Andrea Vedaldi. Visualizing deep convolutional neural networks using natural pre-images. pages 1–23. Springer.

[164] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 2014.

[165] Shobhit Gupta, John A Stamatoyannopoulos, Timothy L Bailey, and William S Noble. Quantifying similarity between motifs. volume 8, page R24. BioMed Central Ltd, 2007.

[166] Fabian A Buske, Mikael Bodén, Denis C Bauer, and Timothy L Bailey. Assigning roles to dna regulatory motifs using comparative genomics. volume 26, pages 860–866. Oxford Univ Press, 2010.

[167] Yijun Xiao and Kyunghyun Cho. Efficient character-level document classification by combining convolution and recurrent layers. 2016.

[168] Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. volume 18, pages 1527–1554, 2006.

[169] Sungchul Ji. The linguistics of dna: words, sentences, grammar, phonetics, and semantics. volume 870, pages 411–417. Wiley Online Library, 1999.

[170] Gary D Stormo. Modeling the specificity of protein-dna interactions. volume 1, pages 115–130. Springer, 2013.

[171] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Advances in Neural Information Processing Systems*, pages 2368–2376, 2015.

[172] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. volume 12, pages 2493–2537. JMLR. org, 2011.

[173] Gary D Stormo. Dna binding sites: representation and discovery. volume 16, pages 16–23. Oxford Univ Press, 2000.

[174] Jifeng Dai and Ying-Nian Wu. Generative modeling of convolutional neural networks. 2014.

[175] Anthony Mathelier, Oriol Fornes, David J Arenillas, Chih-yu Chen, Grégoire Denay, Jessica Lee, Wenqiang Shi, Casper Shyr, Ge Tan, Rebecca Worsley-Hunt, et al. Jaspar 2016: a major expansion and update of the open-access database of transcription factor binding profiles. page gkv1176. Oxford Univ Press, 2015.

[176] Antonio LC Gomes, Thomas Abeel, Matthew Peterson, Elham Azizi, Anna Lyubetskaya, Luís Carvalho, and James Galagan. Decoding chip-seq with a double-binding signal refines binding peaks to single-nucleotides and predicts cooperative interaction. volume 24, pages 1686–1697. Cold Spring Harbor Lab, 2014.

[177] Juho Rousu and John Shawe-Taylor. Efficient computation of gapped substring kernels on large alphabets. *Journal of Machine Learning Research*, 6(Sep):1323–1344, 2005.

[178] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC bioinformatics*, 12(1):333, 2011.

[179] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[180] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC genomics*, 9(1):517, 2008.