Identification Of Prostate Cancer-Associated Genomic Alterations
By Analyzing Variant Frequencies, Functional Effects, And Protein Interactions

Bofei Wang
University of Texas at El Paso

Follow this and additional works at: https://scholarworks.utep.edu/open_etd

Part of the Biology Commons

Recommended Citation
https://scholarworks.utep.edu/open_etd/3459

This is brought to you for free and open access by ScholarWorks@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.
IDENTIFICATION OF PROSTATE CANCER-ASSOCIATED GENOMIC ALTERATIONS BY ANALYZING VARIANT FREQUENCIES, FUNCTIONAL EFFECTS, AND PROTEIN INTERACTIONS

BOFEI WANG

Doctoral Program in Computational Science

APPROVED:

Ming-Ying Leung, Ph.D., Chair

Marc B. Cox, Ph.D.

M. Shahriar Hossain, Ph.D.

Jonathon E. Mohl, Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School
IDENTIFICATION OF PROSTATE CANCER-ASSOCIATED GENOMIC ALTERATIONS BY ANALYZING VARIANT FREQUENCIES, FUNCTIONAL EFFECTS, AND PROTEIN INTERACTIONS

by

BOFEI WANG, M.S.

DISSE SunATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Computational Science Program

THE UNIVERSITY OF TEXAS AT EL PASO

December 2021
Acknowledgements

Throughout the working of this project and writing of dissertation, I received a great deal of support and help. I want to take a moment to thank them.

I would like to first express my sincere gratitude to my supervisor, Dr. Ming-Ying Leung, the director of Computational Science Program at The University of Texas at El Paso, whose expertise was invaluable in designing the research. She always provides insightful suggestions and clear explanations when I was lost. Her patience, encouragement, motivation, immense advice and continuous support brought my work to a higher level. I can never reach this point without enormous help from Dr. Leung.

I also want to thank other members in my committee, Dr. Hossain from Computer Science Department, Dr. Cox from Biological Sciences Department and school of pharmacy and Dr. Mohl from Mathematical Sciences Department, all at The University of Texas at El Paso. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Also, the technical support in computing from Dr. Mohl is highly appreciated. My heartfelt gratitude also goes to the previous and current group members: David Armendariz, Angelica Marquez, Mariana Marquez and Kelvin Minta Ofori for their enormous participation in putting together this work. Additionally, a special gratitude goes to funding sources NIH Grant 5G12RR007592.

Finally, I want to thank my families. My parents gave me huge support for going abroad, studying and living here. During the completion of this work, they supported me spiritually when I was frustrated by difficulties. My fiancée Cuipeng Qiu, has been always there with me, listening to me pouring out my feelings and setbacks I met. She was never fed up but kept comforting me and giving me suggestions in academic and life. I have no words to express my love.
Abstract

With the ever-increasing varieties of sequencing techniques, the volume and scope of genomic data are explosively expanded, offering unparalleled opportunities for researchers to study gene-disease associations, identify biomarkers, and thus develop more effective diagnostic and therapeutic strategies. In this project, I have developed a computational workflow and a new scoring scheme, which combine statistical frequency-based analyses with two well-established functional effect prediction tools FATHMM and PROVEAN, to evaluate nonsynonymous GSVs and identify potential cancer-related protein-coding genes for downstream enrichment and protein-protein interaction (PPI) studies.

This method has been applied to process a collection of 503 whole exome sequencing datasets for patients with prostate cancer (PrCa). The datasets were downloaded from The Cancer Genome Atlas as variant call format (VCF) files containing GSV information for paired tumor and normal samples. Exploratory statistics revealed unusually high level of transitions G→A and C→T among cancer samples. Furthermore, 5 GSVs were found significantly associated with the disease. Among 61 high-scoring genes identified by our scoring scheme, 27 were found by PPI analysis to have degrees of connection ≥ 4 with well-known PrCa-related genes. While 18 of them are reportedly associated with PrCa, 9 genes (TRRAP, EPHB1, HERC2, MCM3, SPTA1, SALL1, HERC1, TTN, and MYH6) have not been previously documented in relation to PrCa. Their potential roles in PrCa could be investigated by further bioinformatics and wet-lab studies.
# Table of Contents

Acknowledgements ......................................................................................... iv

Abstract ........................................................................................................ v

Table of Contents ............................................................................................. vi

List of Tables .................................................................................................. ix

List of Figures .................................................................................................. x

Chapter 1: Introduction .................................................................................... 1
  1.1 Background and significance ................................................................. 1
  1.2 OncoMiner pipeline ............................................................................. 3
  1.3 Research aims ...................................................................................... 4

Chapter 2 Literature Review ........................................................................... 6
  2.1 Prostate cancer ..................................................................................... 6
  2.2 Whole genome sequencing analysis .................................................... 8
    2.2.1 Quality control ............................................................................ 9
    2.2.2 Read alignment .......................................................................... 10
    2.2.3 Variant calling ........................................................................... 10
    2.2.4 Annotation .................................................................................. 12
    2.2.5 Visualization .............................................................................. 12
    2.2.6 Functional effect prediction ....................................................... 13
    2.2.7 Prioritization and filtration ......................................................... 13
  2.3 Available software and programs .......................................................... 14
    2.3.1 Quality control tools .................................................................. 14
    2.3.2 Alignment tools .......................................................................... 15
    2.3.3 Variant valling tools ................................................................... 16
    2.3.4 Annotation tools ......................................................................... 17
    2.3.5 Prioritization and filtration tools ................................................. 17
      2.3.5.1 PolyPhen2 and SIFT .............................................................. 18
      2.3.5.2 FATHMM ............................................................................. 19
      2.3.5.3 PROVEAN .......................................................................... 20
    2.3.6 Resources for mutations, genes and pathways ........................... 21
Chapter 6 Conclusions and future work..................................................................................65
References..................................................................................................................................66
Appendix.......................................................................................................................................75
  A File conversion and variant annotation ..............................................................................75
  B Subject counts ......................................................................................................................94
  C Functional predictions .........................................................................................................99
  D Pathogenic scores calculation ..........................................................................................109

Vita 112
List of Tables

Table 4.1: GSV counts in tumor and normal samples at various types of genomic regions, along with the tumor/normal ratio. ......................................................................................................................... 42
Table 4.2: Base counts on GSV-gene transcripts and observed number of 12 different substitutions on tumor and normal samples. ................................................................................................................................. 45
Table 4.3: Top 10 GSV-genes with the highest $\Delta T$. ................................................................................................................................. 47
Table 4.4: GSVs statistically associated with PrCa (Adjusted $p < 0.05$ in McNemar’s test)...... 48
Table 4.5: Top-scoring pathogenic genes with 4 or more direct interactions with LPC genes. In boldface are those that are also LPC genes themselves. ................................................................................................................................. 55
List of Figures

Figure 1.1: OncoMiner pipeline internal webpage ................................................................. 3
Figure 1.2: Example of OncoMiner input file ........................................................................ 4
Figure 1.3: VCF file example ................................................................................................. 4
Figure 3.1: Overall workflow ........................................................................................................ 26
Figure 3.2: Example of refFlat files used to annotate GSVs .................................................. 28
Figure 3.3: Decision tree used to classify genomic regions for GSVs ..................................... 29
Figure 3.4: Example of FATHMM input file ............................................................................ 32
Figure 3.5: Example of amino acid variation file for PROVEAN .......................................... 33
Figure 3.6: Example of input file for SNPnexus .................................................................... 34
Figure 4.1: Average runtimes for test VCF files using multiple cores .................................... 39
Figure 4.2: Average efficiencies for various cores ................................................................. 40
Figure 4.3: Tumor/normal GSV counts in patient files ........................................................... 42
Figure 4.4: Transition/transversion splits in transcripts, intron and exon among tumor(a) and normal(b) samples ........................................................................................................ 44
Figure 4.5: Mutational levels ($S_{XY} \times 10^6$) in gene transcripts, exon and intron regions .... 46
Figure 4.6: Top 10 terms of GO annotation for BP, CC and MF ........................................... 51
Figure 4.7: KEGG pathway enrichment analysis .................................................................... 52
Figure 4.8: Protein-protein interaction network of 61 high-scoring pathogenic genes .......... 53
Figure 4.9: Interactions between 15 high-scoring pathogenic non-LPC genes (yellow) with LPC genes (blue). Interactions among the blue nodes are not shown ........................................... 54
Chapter 1: Introduction

Next-generation sequencing (NGS) technologies have become indispensable tools for characterizing genomics, transcriptomics and proteomics. As of the beginning of the 21st century, more and more biotechnology companies started to release various DNA sequencing platforms and enormous amounts of data have been generated. At the same time, numerous computational tools and analytical algorithms designed for handling large-scale sequencing data are emerging, which greatly speeded up the research in many fields of biological sciences. In 2016, our research group initiated the implementation of a bioinformatics pipeline named OncoMiner to integrate the analysis of genetic sequence variants (GSVs). Over the past few years, more functional modules have been added to the pipeline in order to generalize and expand the functionalities of OncoMiner with the ultimate goal of establishing a reliable and versatile computational tool for mining NGS data, identify cancer-related genetic sequence variants, and predict their functional effects in order to help reveal potentially deleterious or pathogenic GSVs in various cancer types.

1.1 Background and Significance

Cancer is a complex disease resulting from the accumulation of genetic alterations in the human genome [1]. Although most alterations are considered as passengers and can be present in normal cells, specific patterns of mutations may lead to tumor progressions [2]. Therefore, discovering cancer-related genes and genetic sequence variants (GSVs) would contribute to the establishment of promising tumor biomarkers or targeted therapies, which will improve the tumor detection, prognosis and treatment [3]. In recent years, the rapid revolution in sequencing techniques and increased molecular characterization of tumor tissues have accelerated the development of cancer biomarkers [4].
High-throughput sequencing technologies are yielding extensively large-scale sequencing data. Analyzing these data could help identify genetic alterations in human genome, reveal underlying biological processes and molecular mechanisms that promote cancer development [5]. Numerous bioinformatics and computational tools have been developed to annotate, predict and interpret functional effects of variants. MutsigCV [6] analyzes frequent DNA mutations by building a model assessing the background mutation processes. Polyphen-2 and SIFT are well-established software using naïve Bayes classifier and sequence homology-based method respectively [7,8]. FATHMM creates alignment profile with hidden Markov model and predicts variant impact based on evolutionary features [9]. PROVEAN predicts effect of variations at protein level by calculating the difference between alignment scores after and before the introduction of variants [10]. Other existing function prediction tools include MutationTaster, Mutation Accessor etc. [11,12]. While these programs differ in their underlying algorithmic approaches, most of them classify variants based on information of evolutionary conservation, sequence homology or physicochemical properties of amino acids [13]. In addition, most of these tools were developed before the most recent human genome build (GRCh38) was available and remained not updated, which often limit their generalizability and applications.

Prostate cancer (PrCa) is the most frequently diagnosed cancer in men in over half of the countries in the world. In 2020, an estimation of almost 1.4 million new cases and 375,000 deaths worldwide made it the second most frequent cancer and the fifth leading cause of cancer deaths among men [14]. Prostate-specific antigen test is commonly used for screening PrCa. However, this procedure is painful and often leads to overtreatment [15]. Some blood-based and urine-based biomarkers for PrCa like PCA3 and PTEN have been reported, but their clinical value remain
undetermined [16]. Moreover, given the large set of GSVs or mutated genes, it is time-consuming and expensive to identify the ones associated with PrCa.

1.2 OncoMiner pipeline

A number of online or standalone tools have been developed to analyze NGS data, but few of them can perform the whole NGS workflow. OncoMiner [17] is a bioinformatics pipeline developed at UTEP initially for mining a local set of whole exome sequencing (WES) data. The pipeline was implemented in 2016 as a web server (OncoMiner.utep.edu) to assist biomedical researchers in our Border Biomedical Research Center at UTEP to analyze the datasets obtained from the local population and cell lines. The pipeline is able to identify GSVs, link them with associated research literature, visualize their genomic locations, and compare their occurrence frequencies among different groups of subjects.

The OncoMiner homepage is shown in Figure 1.1. The input data format is limited to a CSV file with specified columns including chromosome, position of GSV, reference base, altered bases, quality score, gene name, genomic region, change type of amino acid (AA), etc. as shown in Figure 1.2. On the submission page, an email address is preferred so that upon completion of processing, users can be notified by an email containing a link to the annotated output file.

Figure 1.1: OncoMiner pipeline internal webpage
More recently, OncoMiner has been generalized to accept more common NGS files in FastQ or BAM/SAM format. The codes for preprocessing FastQ and BAM/SAM files in parallel environments have already been developed by other members in our group and published in 2018 [18]. As part of my work, I further extended the OncoMiner preprocessing (OP) program by developing the functionality to preprocess variant call format (VCF) files (see Figure 1.3 for an example).

![Table](image1.png)  
**Figure 1.2: Example of OncoMiner input file**

![Table](image2.png)  
**Figure 1.3: VCF file example**

### 1.3 Research Aims

The purpose of my research is two-fold. First, we need to establish additional OncoMiner modules for processing and analyzing NGS data in common file formats. Second, we want to develop a computational workflow to identify potential cancer-associated GSVs and genes based on a combination of comparative analyses of mutation frequencies between normal and tumor samples, functional effect predictions, and protein-protein interaction networks. To achieve these
goals, I am using a large collection of VCF files containing WES data from about 500 prostate cancer patients. The specific aims are:

1. Develop python scripts to efficiently convert VCF files to OncoMiner input files and evaluate the performance of the script.

2. Compile GSVs in all VCF files and analyze the overall mutational landscape.

3. Predict functional effects of nonsynonymous GSVs in each protein-coding gene and design a scoring scheme to assess the combined pathogenic effects of the GSVs on the gene.

4. Perform bioinformatics analysis on high-scoring genes selected in aim 3 above in order to reveal their potential involvement in PrCa.

The proposed approach will efficiently process genomic data and reduce the large list of variants and genes by prioritizing candidates that are more likely to be associated with PrCa. These highly prioritized GSVs and genes will be reported so that they can be further investigated by wet-lab studies. The overall workflow is being incorporated into the OncoMiner Pipeline to enable its application in other types of cancer data.
Chapter 2 Literature Review

In this chapter, I will provide background knowledge about the biology of Prostate Cancer (PrCa), whole genome sequence (WGS) techniques, the developed software tools for WGS analysis, and related computational techniques and statistical tools. At the end, I will also review published articles about discoveries of PrCa-related genes, proteins or established biomarkers, etc. Additional materials of DNA sequencing, popular programs for WGS analysis and PrCa genomics can also be found in basic textbooks or published literatures.

2.1 Prostate Cancer

PrCa is one of the most common cancers and is among the leading causes of cancer-related mortality in men. It alone accounts for more than 1 in 5 newly diagnosed cancer patients in 2020 [19]. It occurs in the prostate, a gland in men that produces the seminal fluid that nourishes and transports sperm. The detection, because of the lack of obvious symptoms, is usually based on the abnormal prostate-specific antigen (PSA) level in blood, followed by transrectal ultrasound guided biopsy. However, PSA levels cannot typically reflect advanced stages of diseases [20]. Risk factors for PrCa includes age, race, family history, eating habits et al. Age is closely related to the risk of PrCa. It is rare among men younger than 40 years of age but the incidence rate begins to rise after 55 years of age and reach the peak at around 70 [21].

Earlier studies on the molecular basis for PrCa disparity have focused on the influence of heritable mutations and single-nucleotide polymorphisms (SNPs). Most PrCa susceptibility alleles identified based on genome-wide association studies (GWAS) were common, low-penetrance variants [22]. Since 2007, many PrCa risk-associated SNPs have been identified through genome-wide association studies (GWAS). Till 2018, more than 163 SNPs have been confirmed to be associated with the risk of PrCa [23]. The effect of a single variant is usually modest, so many
studies focus on the cumulative risk of a number of genetic variants. Zheng et al evaluated 16 SNPs from five chromosomal regions (three at 8q24 and one each at 17q12 and 17q24.3) in a Swedish population. They assessed the individual and combined association of the SNPs with PrCa. Five SNPs were found to be weakly associated with PrCa, but their combined effect was much stronger [24]. Apart from these five variants, Helfand et al. also identified 4 additional variants located on chromosomes 2p15, 10q11, 11q13 and Xp11[25]. They assessed both the independent and cumulative risk of all 9 variants and found that PrCa cases had an increased incidence of all 9 risk variants compared to controls or the previous 5 SNPs. Particularly, they found that men with 6 or more variants were at greater than 6-fold increased risk for PrCa.

More recently, advances and general applications of high throughput sequencing technologies have enabled researchers to identify the differences in genome between individuals. Genomic studies enabled by next-generation sequencing (NGS) have focused on the identification of somatic mutations. These variants between normal people and PrCa patients might be the cause of disease. Xu et al sequenced 5 prostate cancer tissues and discovered, in 92 genes, 116 significantly disruptive mutations including frameshift insertions and deletions (indels) and nonsynonymous mutations [26]. Chang et al examined 11,119 human tumors, spanning 41 cancer types and detected a list of likely pathogenic mutations in well-known cancer genes including SPOP, TP53 for PrCa [27]. The identification of more and more variants could provide a better understanding of PrCa tumorigenesis and also help with the prediction of incidence and prognosis. On-going discoveries of somatic mutations and their clustering on genes keep revealing classic and novel PrCa-related genes like SPOP, FOXA1, PTEN, KCNT2 etc. and their roles in the development of PrCa [28, 29]. A detailed review of PrCa driver genes and pathways is covered in section 2.4.
2.2 Whole genome sequencing analysis

Human genome contains around 3 billion base pairs. The order of nucleotides in cells contain the information and instructions for producing proteins and other molecules that is essential for cells to carry out its daily work. Thus, knowledge of DNA sequences plays a key role in understanding the biology of various diseases, reveal the mechanisms of their progression and developing targeted medication. However, pioneering DNA sequencing methods are very time consuming and costly. Since the 21st century, with the advent and rapid development of DNA sequencing technologies, a large amount of sequencing data has been generated more efficiently and with low cost which revolutionized genomic and genetic researches.

There are currently numerous commercial platforms from different companies for NGS. Illumina occupies the largest market share in the field. Its sequencers are able to generate 3 - 13 billion reads in a single run. Other companies including Pacific Biosciences, BGI Genomics, Thermo Fisher Scientific, etc. are also developing efficient platforms with improved throughput, read length and accuracy.

Exome is part of transcribed genome sequence that remains in the mature RNA. It only constitutes less than 2% of the human whole genome, but about 85% of known disease-related mutations were found within this region and mutations on exons were shown to cause the majority of monogenic diseases [30]. Therefore, whole exome sequencing (WES) makes it more efficient and cost-effective to identify associations between GSVs and diseases, which may lead to better clinical diagnosis and treatments. Also, the low load of sequencing also reduces the complexity of data processing, analysis and interpretation. In this section, I will describe classic procedures for analyzing WGS and/or WES data from these platforms.
In general, raw sequencing data directly from sequencers are stored in fastq format. Other types of files used include standard flowgram format (SFF), binary alignment map (BAM) and so on. Fastq is a text-based file using four lines for each sequence. It starts with a '@' character followed by a sequence identifier. The basic structure is the same as Fasta file, but fastq contains one more line specifying the quality score of data. Since the data is unaligned raw data, the pre-processing steps start with quality control (QC), and followed by alignment and post processing, like removal of low coverage and duplicate reads [31].

### 2.2.1 Quality control

NGS data are being generated at an unprecedented pace, but the quality of data and accuracy of analyses could be affected by artifacts from library or sequencing error [32]. Thus, the first step after completing the sequencing run is to assess the quality of raw data and filter out reads that do not meet pre-defined criterions. The early detection of low-quality data may reduce the amount of further downstream analyses and increase accuracy. The quality of sequences is generally measured by a Phred score $Q_{\text{Phred}} = -10 \log_{10} P_e$, where $P_e$ is the estimated probability of error [33]. The scores generally range from 2 to 40 with higher scores corresponding to higher quality in the call. A common practice is to filter out bases with Phred scores below 20, but individuals may set other threshold based on their preferences or research demand. The fastq file already incorporates the quality score. There are also several tools developed to assess the quality of raw data from NGS. Removal of undesired sequences such as adaptors and trimming low quality ends from reads and removal of contamination reads are also important as the alignment of contaminating sequences to a reference genome will result in low alignment quality [33].
2.2.2 Read alignment

After removing undesired sequences and controlling for qualities, the following step is to align reads to a reference genome. Currently, two commonly used sources for the human reference genome assembly are hg38 from University of Santa Cruz (UCSC) and GRCh38 from Genome Reference Consortium (GRC). The difference between two assemblies is tiny the same, but differ in their nomenclature [34]. The mapping of sequenced reads to reference is a classical string match, but due to the sheer amount of NGS data and huge volume of human genome, this process usually involves complicated and computationally intensive work. Alignment results are usually stored in a Sequence Alignment Map (SAM) or its equivalent compressed binary form BAM.

The aligned sequence still need to be processed as some bad features of sequences only show up after alignment. The PCR amplification, which is conducted before alignment to ensure good coverage and depth, may introduce duplicate reads to sequence data, which can affect further downstream analyses [11]. Thus, it is necessary to remove them before the identification of variants. Indels are most likely to be misinterpreted by variant calling algorithms as overlapping reads may have different alignment results in indel regions. So, local realignment of indels using known genetic variants as references are also important for improving accuracy [35,36]. Then base quality score recalibration that recalculate the Phred score needs to be performed to correct for error covariates and reduce false positives [37].

2.2.3 Variant calling

Variant calling is the identification of where aligned reads differ from a reference genome and writing to a Variant Call Format (VCF) file, which is a widely used format for storing variant information. Genetic variants take many forms and according to spectrum of sizes, they can be
broadly classified into three categories: single nucleotide variant (SNV), indel and large structural variant (SV) [38].

SNV is a substitution of one nucleotide by another. Based on the type of nucleotide, it could be further divided into transition (interchange of the purine (Adenine/Guanine) or pyrimidine (Cytosine/Thymine) nucleic acids and transversion (interchange between purine and pyrimidine). Based on the effect of variations, it can be divided into missense (result in a change at amino acid level), nonsense (result in a stop codon that shorten the protein sequence) and silent (no change of amino acid). Short indels refers to a small piece of DNA, usually ranging from 1 to tens of bases that has either been inserted into or deleted from the genome. They are the second most abundant form of human genetic variations [39]. SV typically occurs over a large sequence. This category usually includes copy number variant (CNV), translocation, inversion etc. in the genome that may change the structure of coded proteins. CNV is the duplication or triplication of a stretch of DNA. Inversion is the flip of some region along the genome. Translocation is the exchange of sequences between two non-homologous chromosomes [40]. Long indels are also considered as SVs.

According to the location of mutations, it can also be divided as germline mutations and somatic mutations. Germline mutations occur in gametes. It can be incorporated into the DNA of every cell and passed onto offspring. However, a somatic mutation only occurs in a single body cell and cannot be inherited. It only affects tissues derived from the mutated cell.

Variants are identified by comparing mapped reads to the reference genome. There are numerous variants calling programs that use different algorithms to decrease computational complexity and increase accuracy. Techniques used in these algorithms will be covered in section 2.3. The results of variant calling are stored in a VCF file, which is composed of meta-information
lines, a header line, and data lines. Meta-information lines describe the basic information and data structure of the file. The header line names 8 required fields and may include other optional columns. Each line in data section represents a variant, containing all information listed in the header line.

2.2.4 Annotation

Variant annotation is an important but complex step in the analysis of NGS data. There are many different types of information that could be associated with variants, from measures of sequence conservation to predictions about the effect of a variant on protein structure and function [41]. Many annotation tools have been developed. However, not as convenient as other steps that have been automated, some part of annotation still requires manually review, interpretation, and validation due to the inconsistence between various databases [42]. The most fundamental annotation is categorizing variants based on their genomic regions (transcript, exon, intron, etc.). Information regarding their gene symbol, amino acid change, allele frequency, possible involvement in disease development, clinical significance, predicted functional effects are also attached [43]. Most tools focus on the annotation of SNVs and indels in protein-coding region using information from publicly available database, like dbSNP [44], OncoKB [45], COSMIC [46], ClinVar etc.[47]. Knowledge added to variants will help researchers get a better understanding of variants and their associations with diseases and thus select potentially deleterious ones.

2.2.5 Visualization

The text-based mutation data are not always easy to read and summarize. Transforming such information into a picture would help make sense of distribution features of data and
understand the structure of our data. For example, a bar plot showing the type and abundance of variants, a heatmap comparing variants in different groups or visualizing hotspot mutations [48].

2.2.6 Functional effect prediction

The large-scale variants identified from NGS data create challenges for researchers to look at the pathogenic ones they are interested in as obtaining experimental evidence of pathogenicity of these mutations could be disastrous. Speedy bioinformatics and computational tools have been developed to predict functional effects of variants. These tools (covered in section 2.3) integrate knowledge of amino acid structure, chemical properties, evolutionary conservation etc. to evaluate the pathogenicity of mutations using machine learning or statistical methods [49]. These predictions provide an effective alternative over experimental approaches to prioritize variants of interest and to facilitate the understanding of the mechanisms and associations with disease or pathological phenotypes underlying causal variants [50].

2.2.7 Prioritization and filtration

Selecting variants of interest from a list of thousands to millions of inconsequential variants remains an analytical bottleneck. Filtration for this large candidate list could help prioritize casual variants from common polymorphisms, which may accelerate and simplify variant interpretation. In general, filtration and prioritization include three steps [51]. The first step is removal of less reliable variant or artifacts produced during alignment and variant calling, which includes variants with low quality, strand biased, located in SNV clusters, and/or supported by low-confidence read alignment. The second step is to restrict variants to those of relatively low population frequency, since a huge proportion of human diseases are caused by rare mutations [52]. The third step is to prioritize the variants related to the disease according to their predicted coding-effects. The last step is the most important one as the database-related annotation is limited to known disease and
variants but most variants could remain unknown. Predicted results could help researcher select possibly deleterious variants for further analysis.

2.3 AVAILABLE SOFTWARE AND PROGRAMS

For each step in the analytic flow, plenty of bioinformatics and computational programs have been well established and applied. The underlying algorithms of these tools are different and some comprehensive reviews give detailed comparisons and evaluations of these tools [13, 17, 34, 53]. In this section, I will introduce several popular programs that can be applied to each step mentioned above. Also, some widely used tools incorporating several features and functionalities and complete analytical pipelines will be discussed.

2.3.1 Quality control tools

Most sequencing platforms generate a quality report as part of their functionalities, but usually this report can only identify problems caused by the sequencer itself. Thus, there are some other tools to control the quality of raw sequence data. FastQC (available online at https://www.bioinformatics.babraham.ac.uk/projects/fastqc/) is a classic tool which could provide a wide range of information regarding the quality profile of reads including quality problems originate either in the sequencer or in the starting library material, GC content, over-abundance of adaptors and PCR duplication rate, etc. It accepts files in SAM/BAM or Fastq formats and is compatible with all main sequencing platforms. Newly emerged FQC utilizes FastQC at backend and integrates extended features like plotting and aggregating results of multiple sequence runs into an interactive website [54].

Given the QC report, researchers can determine whether preprocessing steps need to be carried out. AfterQC was designed to profile sequencing errors, analyze the overlapping for pair-end sequencing data and cut adaptors [55]. Cutadapt [56] is capable of finding and removing
adapter sequences, primers, poly-A tails and other types of undesired sequence. Trimmmomatic [57] includes a variety of processing steps for read trimming and filtering, but it is famous for identification and removal of adapters.

### 2.3.2 Alignment tools

Many alignment tools have been established to reveal sequence conservation and variation including Bowtie2 [58], BWA [59], SOAP3 [60], Eland (Illumina suite), NextGenMap [61], rHAT [62], GSAlign [63], Arioc [64] and many others. Among these aligners, BWA, Bowtie (1 and 2) are two of best suited ones for aligning large numbers of relatively short reads and most widely used in NGS analysis.

Bowtie (1 and 2), SOAP (2 and 3), and BWA all use Burrows-Wheeler Transformation techniques. Bowtie2 extends the original ultrafast, memory-efficient short read aligner to allow gapped alignment using dynamic programming. It is also capable of aligning long reads to long genomes. BWA is a short-read aligner which runs slower than Bowtie but allows indels and gaps. The newest version BWA-MEM [65], which was incorporated as a component into BigBWA [66] on a Hadoop cluster, can align long sequences up to megabases and support paired-end reads and perform chimeric alignment. The implementation on Hadoop also substantially reduces computational time, and increase fault tolerance.

Duplicated reads produced by PCR amplification during library construction always show up after alignment. PCR duplicates containing an amplification-induced error may cause a variant calling algorithm to misidentify the error as a true variant [67]. Thus, it is important to remove all but one before calling variants. Picard MarkDuplicates (online version available at http://broadinstitute.github.io/picard/) and SAMtools rmdup [68] are two main software used to remove PCR duplicates. Picard is a set of command line tools written in Java. It identifies read
duplicates (originate from the single DNA fragment) comparing sequences in the 5 prime positions of both reads and read-pairs in a SAM/BAM file and produced a new SAM/BAM file with duplicates marked. SAMtools is a library and software package for parsing and manipulating alignments. It uses similar approaches for duplicate marking or removal as Picard but only keep on read with the best quality. It is also able to convert other alignment formats, sort alignments, call SNPs and small indels and view alignments in text.

2.3.3 Variant calling tools

Some popular variant calling programs for SNV and short indels include SAMtools [68], GATK [69], MuTect [70], FreeBayes [71], Platypus [72] and VarScan2 [73]. GATK, short for The Genome Analysis Toolkit, was originally designed to process sequencing data from Illumina but expanded to handle data from a variety of platforms. It first produces the BAM file using BWA. After some necessary manipulations aforementioned, it calls variants for each sample read, then analyzes variants against known variants, and applies a calibration procedure to compute a false discovery rate for each variant. The most updated version is GATK4, which includes Best Practices workflow that is able to start with the raw sequencing data and provide and extends variant calls into somatic, copy number and structural analyses and offers pipeline scripts for workflows. VarScan2 is a platform-independent mutation caller that detects germline, somatic, copy number and multi-sample variants. It uses a heuristic and statistical algorithm to classify somatic status and detect variants in exome data from tumor–normal pairs according to the read depth, base quality, variant allele frequency, and statistical significance, which reduces the impact of confounding factors. MuTect performs a 3-step strategy which includes preprocessing of noisy reads, a Bayesian classifier for detecting somatic mutations with very low allele fractions, and
post-processing that tune filters to eliminate artifacts. SAMtools, as mentioned above, is also widely used to call variants.

2.3.4 Annotation tools

Tools and packages for annotating variants include but are not limited to ANNOVAR [74], SnpEff [75], SNPnexus [76], SnpSift [77]. ANNOVAR is able to perform annotations of variants from diverse genomes with different levels based on their functional effects, conserved genomic regions, predicted transcription factor binding sites, predicted microRNA target sites and predicted stable RNA secondary structures. It provides both online server wANNOVAR [78] and standalone program which can be run through command line. There different types of annotation supported are gene-based, region-based and filter-based annotations. SnpEff annotates variants based on their genomic locations and predicts coding effects of genes. It contains a pre-built database that supports over 38,000 genomes, but also allows users to build their own database if needed. SnpSift can also annotates genomic variants but is always used together with SnpEff to filter out significant variants. SNPnexus is a web-based variant annotation tool that assists in assist in the selection of functionally relevant mutations. The result contains for queried variants genomic regions, dbSNP identifier (if any), gene/protein consequences, population data, etc.

2.3.5 Prioritization and filtration tools

Tools like VAAST2 [79], VarSifer [80], KGGseq [81], have be developed to filter and prioritize variants. However, these tools rely on information in various databases thus can only predict known variants. Prediction tools like PolyPhen-2 [82], SIFT [8], FATHMM [83], PROVEAN [84] and many others use machine learning or statistical approaches to predict effects of both known and novel variations.
2.3.5.1 PolyPhen2 and SIFT

PolyPhen-2, available as standalone software or via the web server, predicts the damaging effects of amino acid substitutions in human using a naive Bayes classifier. The prediction is based on eight sequence-based and three structure-based features selected by an iterative greedy algorithm. The program takes nonsynonymous variants and return prediction scores ranging from 0 to 1, which suggests the likelihood that a substitution is deleterious. Values close to 0 suggests benign variants while values closer to 1 suggests damaging with a default value of 0.5.

SIFT uses sequence homology and physical properties of amino acid to predict the impact of its substitutions on protein functions, which will help researchers to prioritize variants for further study. The basic idea is that mutations occur at a highly conserved region tend to be deleterious. Given a query sequence, SIFT searches the databases for related protein sequences and obtain an alignment. The multiple alignment of query sequence from PSI-BLAST is then converted into a position specific scoring matrix, which is a l×20 matrix where l is the length of the protein sequence. Each entry \( p_{ca} \), is the probability of AA a at position c of the protein where c ranges from 1 to l and a is any one of the 20 AAs. \( p_{ca} \) is estimated using the following formula [85]:

\[
p_{ca} = \frac{N_c}{N_c + B_c} \cdot g_{ca} + \frac{B_c}{N_c + B_c} \cdot f_{ca}
\]

(2.1)

where \( N_c \) is the total number of sequences in the alignment, \( g_{ca} \) is the sequence-weighted frequency that amino acid a appears at position c in the alignment, \( f_{ca} \) is the pseudo-count that amino acid a appears at position c and \( B_c \) is the total number of pseudo-count.

The score is then normalized on the most frequent AA being tolerated (highest \( p_{ca} \)). It represents the probability that the amino acid change is tolerated. So, the scoring scheme of SIFT is similar to that of PolyPhen-2, but with opposite meaning. A variant with a SIFT score less than the cutoff value (usually set at 0.05) is predicted as deleterious. Because the prediction only
depends on the sequence and does not require the information of protein structures, false positive rate may increase if selected sequences are highly homogeneous. So, the result of SIFT also includes a median sequence conservation value, ranges from 0 to 4.3, to show the diversity of aligned sequences. A warning will appear if the median conservation value is greater than 3.0, which means the confidence of the prediction is low.

2.3.5.2 FATHMM

FATHMM was originally constructed as algorithm that predicts the functional effects of nonsynonymous variants on protein. Homologous sequences were searched to build Hidden Markov Model (HMM) using HMMER3 where sequence conservation is interrogated. SUPERFAMILY and Pfam were searched to identify relevant HMMs. The predicted FATHMM score is essentially a log-odds score as shown in equation 2.2. Later on, pathogenicity weights were incorporated to formulate a weighted scoring (equation 2.3) representing the overall tolerance of proteins. It can be applied to both human and nonhuman species. In an unweighted prediction, the magnitude of the effect is calculated using the following formula:

\[
\text{unweighted} = \ln \frac{P_m/(1-P_m)}{P_w/(1-P_w)}
\]  

(2.2)

where \(P_w\) and \(P_m\) are underlying probabilities for the wild-type and mutant amino acid residues, respectively. For an improvement in human, a weighted prediction is used as follows:

\[
\text{weighted} = \ln \frac{(1-P_w)(W_n + 1)}{(1-P_m)(W_d + 1)}
\]  

(2.3)

where \(W_d\) and \(W_n\) are the pathogenicity weights, representing the relative frequencies of disease-associated and functionally neutral amino acid substitutions mapping onto the relevant hidden Markov model, respectively. Prediction scores close to zero suggest neutral effect of the
variant. However, scores below zero suggest that the substitution is unfavorable and scores greater than zero suggest that a favorable effect of variants.

In the most recent development, FATHMM-MKL uses multiple kernel learning to integrate different types of genomic annotation data into kernel matrices and apply support vector machine to predict effects of both coding and noncoding variants [86]. FATHMM-XF employs the same method but draws on an extended feature set which performs a better accuracy, especially on noncoding regions [9]. In the new scoring schema, predictions are given as p-values in the range [0, 1], where values above 0.5 are predicted as deleterious, and those below 0.5 are predicted to be benign. P-values close to 0 or 1 are the highest-confidence predictions that yield the highest accuracy.

### 2.3.5.3 PROVEAN

PROVEAN is an alignment-based algorithm that predicts the functional impact for all kinds of protein sequence variations (single or multiple AA substitutions, in-frame indels). To make predictions, the program requires the input of query protein sequence and observed protein variations. It measures the change in sequence similarity of a query sequence to its homolog before and after the introduction of an amino acid variation to the query sequence. This change in the alignment score is used as the implication of the effect of sequence variation.

First, PROVEAN searches databases for a set of clusters of homologous and distantly related sequences. Then a delta score, \( \Delta(Q, v, S) \) is defined to represent the change in sequence similarity. \( Q \) is the query protein and \( v \) is the variation with respect to its homologous sequence \( S \). So, for each \( S \):

\[
\Delta(Q, v, S) = A(Q', S) - A(Q, S)
\]
Then, an average delta score is computed over all S. To avoid bias due to over-representation of groups of highly similar sequences in the database, homologous sequences were clustered in advance and average delta score was taken over the representatives of each cluster. The equation is given as follows:

\[
\text{score} = \frac{1}{N} \sum_{c=1}^{N} \left( \frac{1}{N_c} \sum_{i=1}^{N_c} \Delta_{c,i} \right)
\]

where \( N \) is the number of clusters in the supporting set, \( N_c \) is the number of supporting sequences in the \( c \)-th cluster, and \( \Delta_{c,i} \) is the delta score of the \( i \)-th supporting sequence in the \( c \)-th cluster. This average is used as the final PROVEAN score. Low delta scores are interpreted as amino acid variations leading to a deleterious effect on protein function, while high delta scores are interpreted as variations with neutral effect on protein. The default cutoff value is set as -2.5.

Programs introduced in this part are only small portion of currently existed ones. Extensive summaries of functional prediction tools and methods can be found elsewhere [11,87].

### 2.3.6 Resources for mutations, genes and pathways

Aforementioned tools could help researcher select potentially pathogenic variants from a long list of identified variants. It is essential to link these variants to some biological processes or diseases of interest according to information curated in databases. Variants in genes that are totally unlinked to the known genes or pathways are largely neglected. Gene Ontology (GO) terms annotate genes according to their cellular components, molecular functions and biological processes. Pathway analysis may provide detailed functional insight into the connection between genotype and phenotype, which is critical for understanding pathogenesis and thus developing treatments [88,89]. Public databases like KEGG [90], Reactome [91] and many others are widely used resources to understand how a set of genes interact with each other and function together to
accomplish a specific task. Protein-protein interaction network is a good way to explore relationships between different proteins. Different evidence of interactions (experimental, text mining, predictions, etc) are marked with different colors. DAVID [92] is a popular web-based tool to provide functional interpretation of user defined gene list. Databases like DrugBank [93] can provide links between drugs and their targets (variants), which could help with exploring how functional variants affect drug efficacy and adverse effects. OncoKB knowledge base for cancers that annotates the biologic and oncogenic effects and prognostic and predictive significance of somatic molecular alterations. It also provided summarized list of driver genes for various cancer types [45]. ClinVar is an archive that collects, validates and curates evidence of clinical significance for any type of genetic variants [94]. It is a great source to obtain effect of known variants in clinic.

2.3.7 Complete analytical pipeline

Tools or software mentioned above can only perform one or a few closely related steps in the workflow of WGS analysis. But it could be inefficient if one wants to conduct the complete analysis as output from one program may be incompatible for the next tools then users need to make modifications. Fortunately, there are some analytical pipelines that incorporate these tools into a single workflow. Three recently developed genome analysis pipelines NGS-pipe [95], Fastq2vcf [96] and SpeedSeq [97] were discussed in this section.

NGS-pipe is an automated framework for analyzing whole-exome/-genome and transcriptome sequencing data. It incorporates tools for processing sequencing data, detecting SNVs, indels and CNVs. In addition, for RNA-seq data, gene expression levels is estimated. To ensure reproducibility, if any step of the pipeline failed or produced incomplete results, the whole process is suspended and an error message is thrown.
Fastq2vcf employs tools discussed above for each step to process, annotate and analyze sequence variants in a single or parallel computing environment. FastQC and BWA are used for quality control and sequence alignment. After formatting the data from SAM to BAM, a series of post-processing steps are performed by employing Picard and GATK. The SAMtools, GATK and SNVer [98] are invoked to call variants and generate VCF files. Finally, annotation of variants are carried out by ANNOVAR and VEP [99].

SpeedSeq is an open-source platform that accomplishes alignment, variant detection and functional annotation in a very rapid speed. It uses BWA-MEN to align paired-end FASTQ files to the human GRCh37 reference genome. The duplication marking is assigned to Samblaster [100]. Then FreeBayes is run to detect SNVs and indels. LUMPY, SVTyper (A maximum-likelihood Bayesian classification algorithm developed by the same team) and CNVnator are used together to analyze and annotate SVs.

2.4 MOLECULAR PATHOLOGY OF PROSTATE CANCER

Accumulation of genomic alterations are hallmarks of cancer initiation and progression. Over the past years, advances of sequencing technologies have enabled researchers to discover a wide range of recurrent genomic alterations in PrCa. However, understanding the pathology and developing a reliable biomarker is not easy because of complex interactions between multiple genes/proteins, multiple implicated epidemiological factors and advanced patient age at diagnosis [101]. This section briefly discusses genetic and molecular findings of PrCa.

2.4.1 Driver genes

A number of genes have been reported to be implicated in the development of PrCa. SPOP is one of the most frequently mutated genes in PrCa. Recurrent mutations in SPOP define a subtype of PrCa with notable molecular features including increased levels of DNA methylation,
homogeneous gene expression patterns and frequent overexpression of \textit{SPINK1} mRNA [102]. Blattner et al also demonstrated that mutations in \textit{SPOP} serve as a driver event in invasive PrCa in the mouse model by cooperating with the loss of \textit{PTEN}, which is also a key tumor suppressor in prostate cancer [103]. \textit{FOXA1} is a transcription factor that interacts with androgen receptor to regulate prostatic gene expression [104]. Parolia et al found that alterations of \textit{FOXA1} fall into three structural classes that differ in clinical incidence and genetic co-alteration profiles [105]. Moreover, majority of \textit{FOXA1} coding mutations are found in the DNA binding domain, which decreases its DNA binding affinity [106]. \textit{ATM} is a DNA repair gene that found most frequently mutated in lethal cases of PrCa [107]. \textit{NKX3.1} is a tumor suppressor gene, the expression of which was found to be decreased in high-grade prostatic intraepithelial neoplasia and this loss of expression is strongly associated with hormone-refractory disease [108]. \textit{TP53} is the most frequent mutated in various cancer types while in PrCa, it occurs less common. However, \textit{TP53} mutations have been associated with lineage plasticity involving decreased expression of androgen receptor and luminal differentiation genes [109] and combined loss of \textit{TP53} and \textit{RB1} can increase PrCa resistance to therapies [110].

\subsection*{2.4.2 Genomic pathways}

Androgen receptor (AR) signaling is pivotal in the normal development, function and homeostasis of the prostate. An important finding regarding AR is the recurrent fusion of androgen-responsive gene \textit{TMPRSS2} with oncogenic ETS transcription factor family \textit{ERG} or \textit{ETV1} resulting in aberrant expression of \textit{ERG} or \textit{ETV1}, which provides important implications for understanding prostate cancer tumorigenesis [111]. \textit{GATA2} and \textit{FOXA1} serve as particularly essential cofactors in AR signaling in PrCa cells by involving in androgen regulation of the \textit{PSA} gene and functioning as a pioneer transcription factor to bind to highly compacted chromatin [112].
PI3K pathway functions in cell proliferation, cell transformation, protein synthesis, tumor growth, etc. \textit{ATK}, one of the major downstream targets of PI3K, can be activated by PI3K to phosphorylate downstream targets for regulating various cellular functions [113]. The tumor suppressor \textit{PTEN} is an inhibitor for PI3K/AKT signaling pathway by regulating the expression of \textit{HIF-1} and \textit{VEGF}, which results in the inactivation of \textit{AKT} and thus inhibiting tumor growth of PrCa [114]. WNT signaling is a complex pathway that regulates the self-renewal of PrCa cells with stem cell characteristics independently of AR activity. Therefore, Inhibition of WNT signaling has the potential to reduce the self-renewal of PrCa cells and improve the therapeutic outcome [115].

Many genes involved in DNA repair pathway are mutated in PrCa and they occur more commonly in metastatic than localized prostate tumors. Among the most frequently mutated genes are \textit{ATM}, \textit{PARP1}, \textit{BRCA1} and \textit{BRCA2} [109]. Cells evolved a complex signaling machinery to recognize and repair DNA damage. This machinery includes several pathways with complementary and partially overlapping functions. Different types of DNA damage trigger a response from different branches of this complex system. Alteration in DNA repair genes can cause DNA repair defects which may change the DNA damage response and result in genome integrity, instability, followed by PrCa carcinogenesis and progression [116]. Also, findings suggest that DNA repair genes and AR signaling pathway regulate each other. AR signaling can increase expression of DNA repair genes and promotes PrCa radioresistance by accelerating repair of ionizing radiation–induced DNA damage. In turn, one of the DNA repair proteins, PARP1, is an important cofactor for AR transcriptional activity [117]. Such interactions are important for therapies.
Chapter 3 Materials and Methods

In this chapter, I formally describe methods and procedures used to preprocess VCF data, analyze GSVs, evaluate their functional effects, and score pathogenic genes. An overview of workflow is shown in Figure 3.1.

Figure 3.1: Overall workflow

3.1 Data Acquisition and Preprocessing

A total of 503 variant call format (VCF) files provided by The Cancer Genome Atlas project were downloaded through National Cancer Institute’s data sharing platform Genomic Data Common (https://portal.gdc.cancer.gov/). When navigate to this page, we search project name ‘TCGA-PRAD’ and on the result page, choose experimental strategy as wxs, workflow type as Mutect2, and data format as vcf. Then, all 503 VCF files will show up which can be downloaded as controlled data. Each file contains GSV information, including chromosome number, position of the mutation, dbSNP identifier, reference base, mutated base, genotype, and reading depth, etc.
for a paired sample which consists of a primary tumor tissue sample and a normal tissue sample, from the same PrCa patient.

All VCF files were preprocessed and converted into more condensed CSV files for downstream analysis. The preprocessing tool is mainly composed of three components. First, VCF files are read and processed in Python with the package ‘pyvcf’. The first 8 data fields, listed below, were directly extracted by going through each data line in the VCF file:

(i) numeric identifier (var_index)
(ii) chromosome number (chrom)
(iii) starting position of GSV (left)
(iv) ending position of GSV (right)
(v) reference base (ref)
(vi) mutated base (var_seq1)
(vii) mutated base (var_seq2; among var_seq1 and var_seq2, one is the mutated base and the other is same as ref)
(viii) quality score (var_score, fixed at 28)

To ensure high quality of data and reduce possible effects of sequencing error, GSVs with a read coverage less than 5 or individual minor allele frequency less than 0.05 were removed.

In the second step, gene symbols, genomic regions and change type of amino acid for GSVs in protein-coding regions were annotated based on the reference genome sequence GRCh38 and the corresponding refFlat file, which is a tab-delimited text file holding information about gene structure provided by UCSC Genome Browser. The columns in the refFlat file, as shown in Figure 3.2, represent successively

(i) gene name
The position of each variant was compared with genomic regions in refFlat file and the classification was determined according to the decision tree shown in Figure 3.3. Since one gene may have multiple isoforms, the GSV could be annotated with multiple genes and genomic regions. For GSVs located within the coding region, the change type of amino acid was determined by comparing reference and variant AA. Since we only know the position of GSVs, in order to retrieve the codon from the genome sequence, we need to identify the position of GSV on the...
specific codon. Therefore, we counted the number of bases from the first nucleotide in the coding sequencing to the position of GSV. Then divided this number by 3 and recorded the reminder. If the reminder is 0, the GSV occurs at the third position of codon, which means ref codon is formed by the previous two bases in the genome and reference base at this point. If reminder is 1, the GSV occurs at the first position of codon. So, we took the following two nucleotides. And if the reminder is 2, previous and following bases will be used. The mutated codon is determined in the same way but using mutated base instead of reference base. If reference codon and mutated codon code for different AA, the change type is nonsynonymous. Otherwise, it is synonymous.

An additional column indicating the AA variation was also added. This column contains, for each GSV that causes a nonsynonymous change of AA, the reference and mutated AA and the position of this variation regarding the protein sequence. To find the position of the AA variation in the protein sequence, we first obtained total length \( L \) of coding sequences (CDS) from the first translated nucleotide to the current position of GSV. The position of AA change is calculated as

\[
\text{position of AA variation} = \begin{cases} 
\lfloor L/3 \rfloor & \text{if } L \text{ is a multiple of } 3 \\
\lfloor L/3 \rfloor + 1 & \text{otherwise}
\end{cases}
\]

where \( \lfloor x \rfloor \) is the floor of \( x \).

Figure 3.3: Decision tree used to classify genomic regions for GSVs

An additional column indicating the AA variation was also added. This column contains, for each GSV that causes a nonsynonymous change of AA, the reference and mutated AA and the position of this variation regarding the protein sequence. To find the position of the AA variation in the protein sequence, we first obtained total length \( L \) of coding sequences (CDS) from the first translated nucleotide to the current position of GSV. The position of AA change is calculated as

\[
\text{position of AA variation} = \begin{cases} 
\lfloor L/3 \rfloor & \text{if } L \text{ is a multiple of } 3 \\
\lfloor L/3 \rfloor + 1 & \text{otherwise}
\end{cases}
\]

where \( \lfloor x \rfloor \) is the floor of \( x \).
In the third step, GSV information and annotation in all 503 CSV files resulting from step 2 were merged to produce the final OncoMiner input file.

To improve efficiency, the original refFlat and reference sequence files were split by chromosome for parallel processing. Each VCF file was also split before processing. The multiprocessing module was used to achieve parallelization. After splitting the VCF file, variants on various chromosomes can be processed in parallel following the three steps mentioned above. Each core handles one chromosome at a time and moves on to the next chromosome if done. Variants on all chromosomes are joined together to produce the final OncoMiner input file. The preprocessing program was evaluated and optimized with 148 VCF files on a server with 32 cores and 256 GB memory. The runtime behavior was tested using 1, 2, 4, 8, 16, and 24 cores respectively to compare the speedup and efficiency, which are defined as

\[ S_p = \frac{T_1}{T_p} \quad \text{and} \quad E_p = \frac{S_p}{p} \]

where \( T_p \) is the runtime with \( p \) cores.

3.2 Generation and Exploration of GSV Lists

In order to compare GSV occurrences in tumor and normal samples, 3 disjoint sets of GSVs T1N0, T0N1 and T1N1 were generated from preprocessed data using Python3, where T1N0 is the set of GSVs found only in tumor sample but not in normal sample; T0N1 is the set of GSVs found only in tumor sample but not in normal sample; T1N1 is the set of GSVs found in both. Statistics of GSV counts and nucleotide change types were explored for tumor (T1N0+T1N1) and normal (T0N1+T1N1) samples at various levels of genomic regions, including transcript, exon, intron etc.

Given a specific variant V, the subjects can be partitioned into 4 categories, SubT0N0(V), SubT1N0(V), SubT1N0(V), and SubT1N1(V), where:

SubT0N0(V) = set of subjects who do not have variant V in the tumor and normal tissues,
SubT1N0(V) = set of subjects who have variant V in the tumor but not normal tissue,
SubT0N1(V) = set of subjects who have variant V in the normal but not tumor tissue,
SubT1N1(V) = set of subjects who have variant V in both tumor and normal tissue.

Counts in the 4 categories for each GSV always add up to the total number of subjects included in the study.

GSVs were further grouped by genes they belong to. For each specific gene, we counted the number of distinct GSVs that fall on it for the tumor samples and then for the normal samples. In this counting, each distinct GSV is counted only once regardless of how frequently it occurs among the subjects. Alternatively, if we take the occurrence frequency of GSVs among the subjects into consideration (e.g., if the GSV was found in 10 subjects, it would be counted 10 times), we can produce a tally GSV count for the gene. The tally counts for gene \( g \) for the tumor and normal samples will be denoted by \( T(g, \text{tumor}) \) and \( T(g, \text{normal}) \) respectively. We also define the quantity

\[
\Delta T(g) = T(g, \text{tumor}) - T(g, \text{normal})
\]

(3.1)

to be the difference between the observed tally counts of GSVs in gene \( g \) in tumor and normal samples. A high value of \( \Delta T(g) \) indicates that \( g \) is mutated much more frequently in tumor than normal samples, suggesting possible involvement of \( g \) in the disease.

3.3 Mutational levels for different types of nucleotide substitutions

For all GSVs that fell on gene transcripts, we examined the 12 types of nucleotide substitutions \( X \rightarrow Y \) and compared how often they occur. Taking the background base frequencies into consideration, we measure the level of the mutation \( X \rightarrow Y \) using the fraction

\[
S_{XY} = \frac{\text{Number of } X \rightarrow Y \text{ mutations in sample sequence}}{\text{Number of base } X \text{ observed in reference sequence}}
\]

(3.2)
The values of $S_{XY}$ will allow us to compare the different types of GSVs on tumor and normal samples.

3.4 Prediction of Nonsynonymous GSVs and Pathogenic Genes for Cancer Associations

Nonsynonymous GSVs (nsGSVs) were selected from the tumor list. GSV with multiple annotated regions will be considered as nonsynonymous as long as it has one such annotation. FATHMM and PROVEAN were employed to predict functional effects of nsGSVs.

GSV information was extracted from the merged tumor and normal lists to form the input file for FATHMM-XF (latest version that supports GRCh38), which requires chromosome number (string 'chromosome' and numeric number in separate columns), position of mutation, reference and mutated bases and strand (all ones). An example of input file for FATHMM is shown in figure 3.4. The generated input files were submitted to webserver for predictions.

![Example of FATHMM input file](image)

Figure 3.4: Example of FATHMM input file

PROVEAN requires two input files: the variation of AA change and protein sequence. AA change for nsGSVs were already determined in preprocessing step. For the protein sequence, all translated exon intervals of each gene were obtained from refFlat file. These intervals were used to obtain pieces of sequence from reference genome. Then we concatenate these pieces and translate into protein sequence according to the DNA codon table that maps nucleotide triplets into the corresponding AA. From the beginning of concatenated DNA sequence, every three
nucleotides are translated into an AA represented by a single character. Since multiple variants may fall into the same gene, we group their AA variations in a single file and use the corresponding protein sequence. An example of AA variation file for PROVEAN is shown in figure 3.5.

\[
\begin{align*}
R226Q \\
A343V \\
P362L \\
T437M \\
R962W
\end{align*}
\]

**Figure 3.5: Example of amino acid variation file for PROVEAN**

The standalone PROVEAN program was installed on our local high-performance computing (HPC) platform, an IBM UNIX-based system with 2.53 GHz Intel Xeon processors. We submitted all variations to the load sharing facility batch scheduler with high priority using ‘bsub’ command and distributed on 8 computing nodes of HPC with 5 cores, $3 \times 10^6$ kB file size limit and 2-hour runtime limit allowed on each node.

Prediction scores from both tools were parsed according to their respective default value (0.5 for FATHMM and -2.5 for PROVEAN) and incorporated into a supplemental data file listing all the GSVs on protein-coding regions. Among other items, the file contains two columns, one showing whether the GSV is classified pathogenic or benign by FATHMM and the second by PROVEAN. If a variant is predicted as pathogenic by both, we refer to it as pathogenic variant.

An annotation tool SNPnexus [76] was also included to indicate the novelty of GSVs. SNPnexus currently accepts query input data in three different forms (genomic position, chromosomal region or dbSNP id). For input format of our data, we used genomic position form, which requires six data columns, i.e ‘chromosome’, numeric number of chromosome, position of variant, reference base, alternative base and strand. Columns 3 to 5 can be directly extracted from OncoMiner input file as a data frame. ‘chromosome’ and strand (all ones) were generated and
inserted to the dataframe at corresponding column index. Numeric chromosome number was obtained from the ‘chrom’ column in OncoMiner input file by slicing the string using python. The produced files were submitted to webserver, where users are allowed to select reference genome that corresponds to the data. An example of input file for SNPnexus is shown in figure 3.6.

<table>
<thead>
<tr>
<th>Chromosome</th>
<th>1</th>
<th>942451</th>
<th>T</th>
<th>C</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chromosome</td>
<td>3</td>
<td>9810376</td>
<td></td>
<td></td>
<td>GAT</td>
</tr>
<tr>
<td>Chromosome</td>
<td>7</td>
<td>25226951</td>
<td>TA</td>
<td>GTT</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.6: Example of input file for SNPnexus

Besides FATHMM and PROVEAN scores, the occurrence frequencies of the GSV among the subjects in the tumor and normal samples is also an indicator of potential association with PrCa. Let \( s(v, \text{tumor}) \) and \( s(v, \text{normal}) \) denote the numbers of subjects in whom the variant \( v \) is observed. If the difference \( \Delta s = s(v, \text{tumor}) - s(v, \text{normal}) \) is large, it would suggest \( v \) is worth investigating for possible PrCa association. However, to screen out random noise, we only evaluated those nsGSVs that occur in more than 1% of samples using the McNemar’s test [118] with binomial approximation (equation 3.3):

\[
p = \sum_{i=b}^{n} \binom{n}{i} 0.5^i (1 - 0.5)^{n-i} = 0.5^n \sum_{i=b}^{n} \binom{n}{i}
\]  

(3.3)

where \( b \) is the number of GSVs that occur in tumor but not normal tissues and \( n \) is the total number of GSVs that occur in tumor but not normal and occur in normal but not tumor. The calculated p-values were further corrected for multiple comparison using Bonferroni adjustment [119], which multiply the p-value by the number of total tests conducted.

After scoring each individual GSV with FATHMM, PROVEAN, and its occurrence frequencies in our PrCa dataset, one can also comprehensively evaluate the pathogenicity of any given protein coding gene by combining the evaluations of all the GSVs within. We propose two
scoring functions: \( P_t(g) \) and \( P_l(g) \), both of which incorporate the average FATHMM and PROVEAN ranks, as well as \( \Delta s \) for all the GSVs contained in \( g \). Furthermore, \( P_l(g) \) also takes into account the coding length of genes. The scoring procedure consists of the following steps:

(1) Sort all variants according to their FATHMM and PROVEAN scores from the most pathogenic to the least pathogenic respectively and for each pathogenic variant \( v \), take the sum of two ranks denoted as \( \text{RankSum}(v) \).

(2) calculate its RankScore as follows:

\[
\text{RankScore}(v) = 1 - \frac{\text{RankSum}(v)}{2 \times n}
\]

where \( n \) is the number of pathogenic variants.

(3) Group the pathogenic variants by their corresponding genes, which will be called pathogenic genes from here on. For each pathogenic gene \( g \), we calculate the two pathogenic gene scores \( P_t(g) \) and \( P_l(g) \) as follows:

\[
P_t(g) = \sum_{v \in g} \text{RankScore}(v) \times [s(v, \text{tumor}) - s(v, \text{normal})]
\]

\[
P_l(g) = \frac{1}{\ln l(g)} \sum_{v \in g} \text{RankScore}(v) \times [s(v, \text{tumor}) - s(v, \text{normal})]
\]

where \( l(g) \) is the total length of the coding regions for gene \( g \). The value of \( l(g) \) for each gene was obtained from refFlat file by adding the length of each coding exon. If a gene has multiple isoforms, the longest coding region is used. Normalizing \( P_t(g) \) by the logarithmic scale of \( l(g) \) can counteract the effect of long coding sequence on number of GSVs and make the subject counts between genes at similar scale regardless of gene size.

To demonstrate that the scoring functions \( P_t(g) \) and \( P_l(g) \) are effective in recognizing PrCa-related genes, we need to compare the top-scoring genes with those that are well-known to be related to PrCa. For this purpose, we compiled a list of PrCa-related genes from two sources.
COSMIC cancer gene census [46] contains genes possess casual implications in cancers. The whole list was downloaded from the website (https://cancer.sanger.ac.uk/census) and genes related to PrCa were selected. Martinez et al. [120] constructed a compendium of cancer driver genes by running state-of-the-art driver discovery methods. The PrCa driver genes were downloaded from their IntOGen pipeline (https://www.intogen.org/search). On this website, I selected cancer type prostate adenocarcinoma. Then in the box of mutational cancer driver genes, a downloadable table with all PrCa genes is available. Genes from two sources were merged to form our final list of PrCa genes. It should be emphasized here that the genes obtained above are well-documented to be PrCa-related, but it does not imply that all genes outside of this list are necessarily not related to PrCa.

### 3.5 Enrichment and Interaction Analysis of Top-Scoring Pathogenic Genes

Using the scoring functions in (3.5) and (3.6) respectively, 2 sets of top-scoring 1% pathogenic genes were selected. The union of two sets forms the final gene list for bioinformatics analysis. Gene Ontology (GO) and Kyoto Encyclopedia of Genes and Genomes (KEGG) pathway analyses were performed in R using clusterprofile. The code is as follows:

```r
library(clusterProfiler)
library(org.Hs.eg.db)

select_feature = read.table("differentially_mutated_genes.txt", header=F, sep="",)
common = select_feature[1]
entrezid = bitr(common, fromType = "SYMBOL", toType = "ENTREZID", OrgDb = "org.Hs.eg.db")

ego <- enrichGO(OrgDb="org.Hs.eg.db", gene = entrezid$ENTREZID, ont = "BP", pAdjustMethod="BH", pvalueCutoff = 0.05, readable= TRUE)

ekk <- enrichKEGG(gene= entrezid$ENTREZID, organism = 'human', pvalueCutoff = 0.05, pAdjustMethod="BH", qvalueCutoff = 0.05)
```

Enrichment terms are used to illustrate possible pathways and molecular functions these genes may be enriched in. The Search Tool for the Retrieval of Interacting Genes (STRING:
https://string-db.org) was used to construct protein-protein interaction (PPI) networks, which was visualized and analyzed by Cytoscape [121]. On the homepage of STRING, choose multiple proteins on the left panel and put the list of genes/proteins into the box appear in the search box. Once submitted, analysis of network and export options become available on the result page. Another PPI network was constructed using proteins of PrCa and pathogenic genes. In order to analyze interactions between proteins, the interaction file was downloaded and a list of the prominent pathogenic genes that have four or more direct interactions with PrCa genes were picked out. Interactions between PrCa genes were also removed in advance. The refined interaction file was imported into Cytoscape to visualize associations between these pathogenic and their interacting PrCa genes.
Chapter 4 Results

In this chapter, I present all results related to this work. Supplementary files and basic information of dissertation are stored at the following bioinformatics repository: https://datarepo.bioinformatics.utep.edu/getdata?acc=NJHD3ANPB1JW614. All codes can be found at Github repository https://github.com/bofei-wang/ProstateCancer_TCGA_VCF.

4.1 Compiled datasets of GSVs, GSV-GENES, and PrCa genes

The preprocessing script has been successfully implemented. As expected, using multiple cores substantially reduced overall runtime. The average runtime with various cores is displayed in Figure 4.1. As the number of cores increases, the speedup is quite substantial. The highest speedup achieved was 4.001 using 24 cores. The overall efficiencies using multiple cores were also evaluated and shown in Figure 4.2. To compare the observed efficiencies, we set up a baseline efficiency for each core used as 1/p as if no speedup was achieved. Noting the speedup-efficiency trade-off, we decided to process all 503 using 8 cores. The overall runtime was about 40.6 minutes with average runtime=4.85 seconds/file, speedup=3.144, efficiency = 0.393).

After preprocessing, annotation, and compilation of the 503 WES datasets from PrCa patients, a total of 99856 distinct GSVs were collected of which 99761 were found in tumor samples and 4017 in normal, with 3922 in both. We have put together a large data file (S1.GSV_distribution.csv) with 99856 rows (GSVs) by 1021 columns organized as follows:

(i) The first five columns (A – E) contain chromosome number (chrom), start (left) and end (right) positions of GSV, and the reference (ref_seq) and mutated (alt) bases.

(ii) The next four columns (F – I) contain subject counts SubT1N1, SubT1N0, SubT0N1, SubT0N0. For a GSV in any particular row, these four counts respectively indicate how many subjects have this variant in both tumor and normal samples, exclusively in the tumor sample,
exclusively in the normal sample, and in neither. These four counts always add up to the total number of subjects.

(iii) Columns J – L were obtained using SNPnexus showing the dbSNP identifier (if exists), overlapped gene and genomic region annotation.

(iv) Columns M – O were obtained from our preprocessing script showing the gene name, genomic region and change type of amino acid if the GSV occurs in protein-coding region.

(v) The remaining 1006 columns indicate the presence (1) or absence (0) of each GSV in the tumor and normal samples among the 503 files.
The total collection of GSVs in the samples fell on 17036 genes (S2. GSV_gene_distribution.csv), which will be referred to as GSV-genes from here on. All of them contain some GSVs from the tumor samples, but only 3120 are involved in normal samples. The number of GSVs within each of these gene was recorded for the tumor and normal samples in each patient and the difference was calculated. This generated a matrix with 17036 rows (genes) by 1520 columns organized as follows:

(i) The first three columns (A – C) contain chromosome number, gene symbol and indicator showing whether it is a protein coding gene (Yes) or not (No).

(ii) The next four columns (D – G) contain distinct and tally counts for tumor and normal samples.

(iii) Columns (H – K) contain numbers of GSVs in coding regions and nsGSV for tumor and normal samples.
(iv) The remaining 1509 columns show the count of GSVs in tumor and normal samples among the 503 files and the difference between each pair of samples.

Among the GSV-genes, 15943 are protein coding, all of which contain variants in the tumor samples but only 2981 have variants in normal samples. Supplementary file S3.predict_GSV_CDS.xlsx contains a listing of 33369 GSVs in our dataset that fall on protein coding regions. Among these, only 11 GSVs were found in normal but not in tumor samples. All of the remaining 33358 GSVs, which include 9490 synonymous, 22430 nonsynonymous, and 1438 nonsense substitutions, occur at least once in the tumor samples. More detailed descriptions of the contents in S3 will be provided in later sections.

A total of 82 PrCa driver genes were obtained from Integrative OncoGenomics. Another set of 28 PrCa-related genes was obtained from COSMIC cancer gene census. The union of the two sets forms a list of 102 PrCa genes (see ‘Listed PrCa genes’ tab in S3), which is used in this research to assess the effectiveness of different scoring schemes in selecting PrCa-related genes. In the rest of this dissertation, any gene within this list is called a ‘listed PrCa gene’ (LPC gene) and any gene outside this list a ‘non-LPC gene’. We reiterate here that a non-LPC gene only means that, to date, there is not a sufficient number of studies in the published literature confirming its association with PrCa, but does not imply that the gene is necessarily not PrCa related.

4.2 MUTATIONAL LANDSCAPE

4.2.1 Overview

From the compiled supplementary data files, we can obtain summary statistics of GSV counts considered individually or grouped by genes. Figure 4.3 displays the GSV counts of tumor and normal samples in each patient file sorted by tumor counts with log transformation. However, y-axis shows the original values.
4.2.2 Different types of genomic regions

Table 4.1 below summarizes distinct as well as tally GSV counts in different types of genomic regions. The distinct count simply holds the distinct number of GSVs at each level (number of rows in data) while the tally count takes into consideration the number of subjects with each GSV. For both tumor and normal samples, almost 98% of the GSVs were within gene transcripts regions. Within transcripts, the percentages of GSVs on exons are much higher in tumor than normal samples (41% versus 23%). Similar differences between tumor and normal samples were observed for the percentages of GSVs in protein coding regions, and of GSVs causing nonsynonymous codon changes.

Table 4.1: GSV counts in tumor and normal samples at various types of genomic regions, along with the tumor/normal ratio.

<table>
<thead>
<tr>
<th></th>
<th>Distinct GSV counts (d)</th>
<th>Tally GSV counts (t)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tumor</td>
<td>Normal</td>
</tr>
<tr>
<td>Filtered dataset</td>
<td>99761</td>
<td>4017</td>
</tr>
<tr>
<td>Intergenic</td>
<td>2180</td>
<td>119</td>
</tr>
</tbody>
</table>

Figure 4.3: Tumor/normal GSV counts in patient files
<table>
<thead>
<tr>
<th>Category</th>
<th>Count 1</th>
<th>Count 2</th>
<th>% of Count 1</th>
<th>Count 3</th>
<th>Count 4</th>
<th>% of Count 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene transcripts</td>
<td>97581</td>
<td>3898</td>
<td>25.03</td>
<td>101285</td>
<td>4584</td>
<td>22.10</td>
</tr>
<tr>
<td>Introns</td>
<td>57311</td>
<td>3012</td>
<td>19.03</td>
<td>60038</td>
<td>3606</td>
<td>16.65</td>
</tr>
<tr>
<td>Exons</td>
<td>40270</td>
<td>886</td>
<td>45.45</td>
<td>41247</td>
<td>978</td>
<td>42.17</td>
</tr>
<tr>
<td>Untranslated</td>
<td>6912</td>
<td>348</td>
<td>19.86</td>
<td>73606</td>
<td>398</td>
<td>18.36</td>
</tr>
<tr>
<td>Protein coding</td>
<td>33358</td>
<td>538</td>
<td>62.00</td>
<td>33941</td>
<td>580</td>
<td>58.52</td>
</tr>
<tr>
<td>nonsynonymous</td>
<td>23868</td>
<td>387</td>
<td>61.67</td>
<td>24281</td>
<td>417</td>
<td>58.23</td>
</tr>
</tbody>
</table>

### 4.2.3 Mutations in tumor samples tend to become A/T-rich

There are 12 possible types of single nucleotide substitutions, including the four transitions (A↔G, C↔T) and eight transversions (A↔C, A↔T, C↔G, G↔T). As shown in Figure 4.4, within gene transcripts of all GSV-genes in the tumor samples, transitions constituted 56.1% of all GSVs and this percentage increased further to 65.4% in exons. The split between transitions and transversions was quite even: 49.6% vs. 50.4% in introns. In contrast, transitions made up only 38.2% in the normal samples, and remain around the same, 34.5% and 39.3%, in exons and introns respectively.
The difference in transition percentages between the tumor and normal samples prompted us to examine the 12 types of substitutions more closely. In the tumor samples, the transitions G→A, C→T, were most overrepresented, forming respectively 22.7% and 19.8% of all GSVs. However, other two transition A→G, and T→C were actually below average in frequency. In
normal samples, no such overrepresentation was observed, but A↔T substitutions were highly underrepresented in both directions.

Noting that if mutation occurs randomly, a base X with higher frequency in the sequence would be expected to have a higher number of mutations changing it to a different base. The observed number of the 12 types of substitutions in the tumor and normal samples are given in the Table 4.2 below. From these, we obtain the mutation levels $S_{XY}$ as shown in Figure 4.5.

**Table 4.2: Base counts on GSV-gene transcripts and observed number of 12 different substitutions on tumor and normal samples.**

<table>
<thead>
<tr>
<th>Total base count</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
<th>A</th>
<th>T</th>
<th>C</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>418377428</td>
<td>3013</td>
<td>1953</td>
<td>6611</td>
<td>60</td>
<td>435</td>
<td>379</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>419843455</td>
<td>2946</td>
<td>6623</td>
<td>2192</td>
<td>50</td>
<td>-</td>
<td>348</td>
<td>543</td>
</tr>
<tr>
<td>C</td>
<td>298369704</td>
<td>15810</td>
<td>19321</td>
<td>-</td>
<td>2647</td>
<td>342</td>
<td>412</td>
<td>-</td>
</tr>
<tr>
<td>G</td>
<td>299045605</td>
<td>22197</td>
<td>11728</td>
<td>2540</td>
<td>-</td>
<td>350</td>
<td>390</td>
<td>266</td>
</tr>
</tbody>
</table>

From the mutation levels in the Figure 4.5, it can be seen that

1. Overall mutation levels for all 12 substitutions are higher in tumor than in normal samples.

2. In tumor samples, the strong bases C and G have much higher mutation levels than the weak bases A and T. While mutation levels are much lower in normal samples, similar mutation level differences among the strong and weak nucleotides are observable in the normal samples but to a much less extent.

3. In tumor samples, both C and G have higher tendency to mutate to A or T than to their own complementary bases. This preference is not obvious in normal samples.

4. Mutational levels in exon are extremely higher than that in intron for both tumor and normal samples.
Figure 4.5: Mutational levels ($S_{XY} \times 10^6$) in gene transcripts, exon and intron regions
4.2.4 Tally counts of GSVs reveal highly mutated GSV-genes

For each gene, the tally count of GSVs on a gene reflects the abundance of mutations on it. We use $\Delta T$ to measure the difference of tally counts between tumor and normal samples. The value of $\Delta T$ ranges from 0 to 279 over the collection of GSV genes. The top ten with highest $\Delta T$ are listed in Table 4.3 below.

<table>
<thead>
<tr>
<th>Chr</th>
<th>Gene symbol</th>
<th>Tumor count</th>
<th>Normal count</th>
<th>$\Delta T$</th>
<th>Gene functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chr19</td>
<td>KIR2DS3</td>
<td>291</td>
<td>12</td>
<td>279</td>
<td>Regulation of the immune response</td>
</tr>
<tr>
<td>Chr7</td>
<td>TCAF2</td>
<td>195</td>
<td>3</td>
<td>190</td>
<td>Negatively regulates the plasma membrane cation channel TRPM8 activity; Promotes prostate cancer cell migration stimulation in a TRPM8-dependent manner</td>
</tr>
<tr>
<td>Chr5</td>
<td>PCDHA1</td>
<td>136</td>
<td>1</td>
<td>135</td>
<td>Establishment and function of specific cell-cell connections in the brain</td>
</tr>
<tr>
<td>Chr6</td>
<td>SYNE1</td>
<td>115</td>
<td>3</td>
<td>112</td>
<td>Associated with autosomal recessive cerebellar ataxia; transmission of mechanical forces across the nuclear envelope</td>
</tr>
<tr>
<td>Chr2</td>
<td>TTN</td>
<td>103</td>
<td>0</td>
<td>103</td>
<td>Adhesion template for the assembly of contractile machinery in muscle; kinase activity</td>
</tr>
<tr>
<td>Chr1</td>
<td>CROCCP2</td>
<td>101</td>
<td>3</td>
<td>98</td>
<td>Pseudogene</td>
</tr>
<tr>
<td>Chr21</td>
<td>TPTE</td>
<td>103</td>
<td>6</td>
<td>97</td>
<td>Signal transduction pathways of the endocrine or spermatogenic function of the testis</td>
</tr>
<tr>
<td>Chr1</td>
<td>OBSCN</td>
<td>102</td>
<td>6</td>
<td>96</td>
<td>Organization of myofibrils during assembly; mediate interactions between the sarcoplasmic reticulum and myofibrils</td>
</tr>
<tr>
<td>Chr19</td>
<td>MUC16</td>
<td>90</td>
<td>0</td>
<td>90</td>
<td>Form a barrier to protect epithelial cells</td>
</tr>
<tr>
<td>Chr5</td>
<td>PCDHGA2</td>
<td>85</td>
<td>0</td>
<td>85</td>
<td>Establishment and maintenance of specific neuronal connections in the brain</td>
</tr>
</tbody>
</table>

4.3 Predicting PrCa-associated nsGSVs

As exonic nonsynonymous mutations regions are more likely to cause biological changes associated with diseases [122], we focused on 22430 nsGSVs that occur at least once in the tumor
samples to analyze whether they are benign or pathogenic using two functional prediction tools FATHMM and PROVEAN, which predict variant effects by multiple kernel learning and difference between alignment scores respectively. The prediction results were added to corresponding GSVs in supplementary file S3.

4.3.1 FATHMM and PROVEAN prediction

For those 22430 nsGSVs, 11152 were predicted pathogenic by FATHMM, 12352 by PROVEAN, and 8474 by both with their respective default cutoff values 0.5 and -2.5. We will refer to these 8474 nsGSVs as pathogenic, of which 5238 were found novel when checked against the known GSVs in dbSNP.

4.3.2 Five GSVs showed statistical significance by frequency-based prediction

There were only 11 nsGSVs that occur in 6 or more subjects (> 1% of 503). Comparing their occurrence frequencies in the tumor and normal samples, five nsGSVs showed statistically significant associations with PrCa (Table 4.4) based on the McNemar test.

<table>
<thead>
<tr>
<th>Chr</th>
<th>Position</th>
<th>Gene</th>
<th>FATHMM</th>
<th>PROVEAN</th>
<th>Adjusted p</th>
<th>dbSNP</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>100956292</td>
<td>MUC3A</td>
<td>--</td>
<td>0.781</td>
<td>0.0013</td>
<td>rs1162418239</td>
</tr>
<tr>
<td>10</td>
<td>50068136</td>
<td>WASHC2A</td>
<td>0.0561</td>
<td>-1.601</td>
<td>0.0215</td>
<td>Novel</td>
</tr>
<tr>
<td>15</td>
<td>82434465</td>
<td>GOLGA6L9</td>
<td>0.0530</td>
<td>--</td>
<td>0.0430</td>
<td>rs1367026482</td>
</tr>
<tr>
<td>17</td>
<td>41026893</td>
<td>KRTAPI-5</td>
<td>0.0124</td>
<td>2.800</td>
<td>0.0215</td>
<td>rs746834174</td>
</tr>
<tr>
<td>17</td>
<td>49619070</td>
<td>SPOP</td>
<td>0.9438</td>
<td>-11.628</td>
<td>0.0430</td>
<td>rs1057519968</td>
</tr>
</tbody>
</table>

4.4 Pathogenic GSV-genes possibly associated with PrCa

The 8474 pathogenic nsGSVs span 5357 genes, which will be referred to as pathogenic genes. These pathogenic genes cover 63 LPC genes. Most pathogenic genes contain only one single pathogenic nsGSV but the number goes up to 37. The three genes with the highest count of distinct pathogenic nsGSVs were TTN (37), TP53 (27) and SPOP (23). TP53 and SPOP are well-known cancer driver genes in PrCa [102,123]. TTN was reported to be frequently mutated in
various cancers including PrCa. However, TTN is rarely indicated as a tumor-associated gene [124]. The high mutation frequency may largely due to its giant size. This finding inspires us to take the length of genes into consideration and design a scoring function that incorporates the average FATHMM and PROVEAN ranks, tally counts of GSVs and the coding length of genes. A supplementary file S4.pathogenic_genes.xlsx contains all pathogenic genes and specific aforementioned information including:

(i) Number of distinct pathogenic nsGSVs in this gene.
(ii) Number of distinct coding GSVs in this gene.
(iii) \( \Delta T \) score.
(iv) Length of coding sequencing.
(v) \( P_t(g) \) and \( P_l(g) \) rank.
(vi) \( P_t(g) \) and \( P_l(g) \) rank
(vii) The last 2 columns indicate whether the gene is contained in the the two sources of LPC genes.

According to two scores given by equations (3.5) and (3.6), we selected 1% of pathogenic GSV-genes from each respectively and took the union of 2 sets which resulted in 61 top-scoring pathogenic genes contained in the ‘high_patho’ page of Supplemental file S4. Among the top-scoring pathogenic genes, 12 are LPC genes.

4.5 BIOINFORMATICS ANALYSIS

The final gene list selected for bioinformatics analysis consists of 61 high-scoring pathogenic genes selected by the scoring functions \( P_t(g) \) and \( P_l(g) \) as defined in (3.5) and (3.6).
4.5.1 GO analysis

According to the GO term annotations, these 61 high-scoring pathogenic genes were enriched in 327 terms in biological process (BP), 50 terms in molecular function (MF) and 50 terms in cellular component (CC). The top 10 terms in each category are shown in Figure 4.6. The number at the end of each shows the gene ratio (observed/background) of each term. This result indicated that these genes were mainly involved in the heart and muscle development, transmembrane channel activity, and muscle contraction, etc.

4.5.2 Pathway enrichment

Enriched pathways in Figure 4.7 revealed that the high-scoring pathogenic genes were primarily involved in cardiac functions and diseases, focal adhesion, signaling pathways, and various cancers including PrCa.

4.5.3 Protein-protein interaction networks

The PPI network constructed for the 61 high-scoring pathogenic genes (see supplementary file S4) is shown in Figure 4.8. The network contains 61 nodes and 122 edges with average node degree 4. In the network involving pathogenic genes and LPC genes, we found 27 genes (see Table 4.5) interacting with 4 or more LPC genes, where 15 are non-LPC genes. These 15 pathogenic genes were picked out and their interactions with LPC genes were visualized in Figure 4.9. Table 4.5 lists 27 pathogenic genes that have four or more direct interactions with LPC genes.
Figure 4.6: Top 10 terms of GO annotation for BP, CC and MF
Figure 4.7: KEGG pathway enrichment analysis
Figure 4.8: Protein-protein interaction network of 61 high-scoring pathogenic genes
Figure 4.9: Interactions between 15 high-scoring pathogenic non-LPC genes (yellow) with LPC genes (blue). Interactions among the blue nodes are not shown.
Table 4.5: Top-scoring pathogenic genes with 4 or more direct interactions with LPC genes. In boldface are those that are also LPC genes themselves.

<table>
<thead>
<tr>
<th>Gene</th>
<th>Interaction</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP53</td>
<td>66</td>
<td>LPC gene</td>
</tr>
<tr>
<td>CTNNB1</td>
<td>43</td>
<td>LPC gene</td>
</tr>
<tr>
<td>SMAD4</td>
<td>33</td>
<td>LPC gene</td>
</tr>
<tr>
<td>KMT2D</td>
<td>29</td>
<td>LPC gene</td>
</tr>
<tr>
<td>ATM</td>
<td>28</td>
<td>LPC gene</td>
</tr>
<tr>
<td>KMT2C</td>
<td>27</td>
<td>LPC gene</td>
</tr>
<tr>
<td>IDH1</td>
<td>21</td>
<td>LPC gene</td>
</tr>
<tr>
<td>FOXA1</td>
<td>19</td>
<td>LPC gene</td>
</tr>
<tr>
<td>SPOP</td>
<td>18</td>
<td>LPC gene</td>
</tr>
<tr>
<td>NKK3-I</td>
<td>14</td>
<td>Prostatic tumor suppressor gene, downregulated or loss of expression in advanced PrCa [108,125]</td>
</tr>
<tr>
<td>CSMD3</td>
<td>13</td>
<td>Highly mutated and differentially methylated in PrCa [126]</td>
</tr>
<tr>
<td>TRRAP</td>
<td>12</td>
<td>Potential tumor-suppressive role in breast cancer [127]; regulate TP53 in lymphoma [128]</td>
</tr>
<tr>
<td>BRAF</td>
<td>11</td>
<td>LPC gene</td>
</tr>
<tr>
<td>CHD4</td>
<td>10</td>
<td>Co-repress early growth response activities that are involved in the progression of PrCa [129];</td>
</tr>
<tr>
<td>VWF</td>
<td>9</td>
<td>Increased plasma concentrations in various cancer types [130]</td>
</tr>
<tr>
<td>LRP1B</td>
<td>6</td>
<td>LPC gene</td>
</tr>
<tr>
<td>EPHB1</td>
<td>6</td>
<td>Overexpression associated with a favorable prognosis for glioma; tumor promoter in medulloblastoma [131]</td>
</tr>
<tr>
<td>HERC2</td>
<td>6</td>
<td>Promising drug target for immunotherapeutic treatments in breast cancer development and pathogenesis [132]; mutations detected in leukemia cells, gastric and colorectal carcinomas [133]</td>
</tr>
<tr>
<td>MCM3</td>
<td>5</td>
<td>Independent prognostic factor for hepatocellular carcinoma [134]</td>
</tr>
<tr>
<td>SPTA1</td>
<td>4</td>
<td>Mutated in all stages of small cell lung cancer progression [135]</td>
</tr>
<tr>
<td>SALL1</td>
<td>4</td>
<td>Tumor suppressor in breast cancer [136]</td>
</tr>
<tr>
<td>HERC1</td>
<td>4</td>
<td>High level of expression leads to greater survival in kidney cancer, head and neck cancer, and pancreatic cancer [133]</td>
</tr>
<tr>
<td>RYBP</td>
<td>4</td>
<td>A deleted region harboring RYBP and other genes in PrCa suggests cooperative suppressive role [137]</td>
</tr>
<tr>
<td>TTN</td>
<td>4</td>
<td>Frequently mutated in various cancers; rarely reported to be functionally associated with cancers [124,138]</td>
</tr>
<tr>
<td>CHD5</td>
<td>4</td>
<td>Tumor suppressor: mutated, deleted or silenced in a number of malignancies [139]</td>
</tr>
<tr>
<td>MYH6</td>
<td>4</td>
<td>Cardiac marker gene, sick sinus syndrome susceptibility gene [140]</td>
</tr>
<tr>
<td>FAT3</td>
<td>4</td>
<td>LPC gene</td>
</tr>
</tbody>
</table>
4.6 Python scripts and supplementary files

Four major python scripts have been developed to perform the following tasks in this research:

A. Convert VCF files into CSV file and annotate variants with gene names, genomic regions and change type of AA. This is achieved in parallel processing by splitting VCF files according to chromosomes and distributing the work of extraction and annotation on subfiles to different cores.

B. Merge GSVs from all samples into a single list, construct the binary matrix that mark the GSV occurrences in each sample, and then calculate their subject counts.

C. Select nsGSVs and generate input for FATHMM and PROVEAN; parse prediction results and add them to the GSV list.

D. Group GSVs by genes, obtain CDS length of each gene and calculate their pathogenic scores, and then select top-scoring genes for bioinformatics analysis.

All the above codes are publicly available at Github repository

https://github.com/bofei-wang/ProstateCancer_TCGA_VCF.

These codes are also being incorporated into the next version of OncoMiner to be released at the OncoMiner website (oncominer.utep.edu).

For the various steps in our data analysis, four supplementary files S1.GSV_distribution.csv, S2.GSV-gene_distribution.csv, S3.predict_GSV_CDS.xlsx, and S4.pathogenic_genes.xlsx were organized and they are available at our bioinformatics repository


1. S1 contains information of all distinct GSVs collected from 503 subjects with a binary matrix indicating their occurrences in each patient’s tumor and normal samples.
2. S2 contains information of all GSV-genes, various levels of GSV counts, $\Delta T$, 2 columns indicating the LPC genes and a matrix indicating the number of GSVs on that gene row in the tumor and normal samples together with the difference in between for each subject.

3. S3 comprises two sheets. The first one contains information of coding GSVs and prediction results from FATHMM. AA change and PROVEAN predictions for nsGSVs are also included. The last column indicates whether this GSV is predicted as pathogenic by FATHMM (F), PROVEAN (P), both or neither. The second sheet contains PrCa-associated genes compiled from COSMIC and reference [120].

4. S4 also has two sheets. The first one contains information of all pathogenic genes and two pathogenic scores for each gene, etc. The second sheet contains highly pathogenic genes selected based on two scoring functions and the number of PrCa genes they interact with.
Chapter 5 Discussions

Rapid development and extensive use of high-throughput sequencing technologies in genomics make it especially important to analyze those big data computationally. Comparing with existing prediction tools, the input files for MutSig include a mutation table in the format of Mutation Annotation File (MAF), a coverage table indicating the number of nucleotides sequenced for mutation calling in each patient and each gene and a covariates table that contains expression level and replication time of each gene. However, converting VCF files to MAF are not always easy and one usually does not have information for ‘categ’ required in MAF and coverage files. Polyphen, SIFT and MutationTaster use GRCh37 as reference genome which does not match the one in our data (GRCh38). Mutation Assessor cannot accept VCF files. Therefore, to the best of our knowledge, the proposed workflow is first approach to process a batch of VCF files and comparatively analyze cancer-related GSVs.

5.1 Preprocessing

In this study, we use computational approaches to analyze GSVs and related genes from a large cohort of PrCa patients. The original dataset was preprocessed to extract necessary GSV information and each GSV was annotated with corresponding gene name, genomic region. Parallelization was achieved by splitting data by chromosome and distributing work to different cores. The best speedup achieved was less optimal probably due to several reasons. First, according to Amdahl's law [141], the serial part in program would limit the speedup to the reciprocal of the fraction of serial portion in the code. When checking the runtime using 1 core, the serial proportion was about 17.5% of the overall runtime, which limited the best speedup at 5.7. Second, I/O issues and the communications between independent processes can also increase the runtime. Therefore, the observed speedup was below the ideal value. Finally, a decrease in efficiency was observed
with increasing number of cores. This is because not all cores were fully utilized during the whole process when using multiple cores as different chromosomes contain different numbers of GSVs.

5.2 Mutational landscape and discriminating GSVs

At the variant level, tumor sample was predominantly composed of transitions and the percentage further went up when restricted to exonic regions. This is largely due to the extremely high level of transitions $G \rightarrow A$, $C \rightarrow T$ and such observations were also reported in other cancer types [142,143]. In addition, transversions $G \rightarrow T$, $C \rightarrow A$ also showed much higher levels in tumor samples, which also stands when the background base frequencies were taken into consideration, suggesting their possible impact in cancer development. As previous studies have suggested that DNA methylation at CpG sites could cause mutations leading to genetic disease [144], it would be of interest to further investigate the mutations on G and C bases and their possible ties to cancer.

The McNemar test revealed five discriminating nsGSVs (see Table 4.4) that showed significantly higher occurrence frequencies in tumor than normal samples. In particular, the one at position 50068136 on chromosome 10 within the WASHC2A gene has not been previously reported in the dbSNP database.

Another noteworthy nsGSV, designated rs1057519968 and located at base 49619070 on chromosome 17, is predicted to be highly pathogenic by both FATHMM and PROVEAN. It falls on position 20187 of the famous PrCa driver gene SPOP and caused the W131G mutation in the protein. This consistency across different predictions strongly supports the involvement of this GSV in PrCa. It is also curated as ‘likely pathogenic’ in ClinVar [27], but how this mutation contributes to the disease remains to be understood.
5.3 Functional Effect Predictions

Functional predictions by FATHMM and PROVEAN revealed 8474 possibly deleterious nsGSVs. Among nsGSVs, 311 GSVs did not get predicted by FATHMM. These include 10 GSVs on chromosome X which were not predicted because FATHMM was designed to work on autosomal chromosomes only. The explanation for the remaining ones may come from the minor gap between different patch of GRCh38. Running PROVEAN on HPC is burdensome due to the large number of AA variations and size of protein sequences. The work was submitted with ‘bsub’ but memory needed to be released every 1000-2000 proteins. Variations on some giant proteins were manually submitted to the online version as they reached the maximum memory allowed on our local server. It should be noted that other existing traditional tools like Polyphen and SIFT were not used in our analysis as they were based on an earlier version of reference genome GRCh37, thus not applicable to our data.

From the compiled gene lists, top 10 genes were identified based on their ΔT scores (i.e., the occurrence frequency difference between tumor and normal tissues across all patients). In line with published literatures, SYNE1 and MUC16 were found to be frequently mutated in PrCa [123]. Mutations in SYNE1 may also increase the risk of invasive ovarian cancer [145]. MUC16 encodes a high molecular weight glycoprotein which poses a high risk of residue change. It is normally expressed by epithelium reproductive organs and is a well-known serum marker for ovarian cancer [146]. Studies have shown that overexpression of MUC16 induced proliferation, migration and invasion of epithelial ovarian cancer through the PI3K/AKT signaling pathway [147] and p120-catenin cytoplasmic translocation [148]. Lakshmanan et al. [149] found that it regulates TSPYL5, leading to decreased tumor suppressor activity of p53, which in turn promotes the lung cancer cell growth. Many studies have also observed deregulation of other members of mucin family in
various cancer types [146,150]. Aberrant expression of tumor-associated mucins induces cell invasion, migration and extravasation via different pathways.

*OBSCN* and *TTN* were identified as highly mutated genes in various cancer types, especially the breast cancer [151,152], which shares some cancer driver genes with PrCa, like *BRCA1* and *BRCA2*. Mutational event in *OBSCN* were implicated to affect GPCR, Ras, p75 and Wnt signaling pathways which regulate cell proliferation and differentiation in breast cancer [151].

*TPTE* is a cancer-testis antigen expressed only in tumor tissue not normal testis. It is a homolog of tumor suppressor gene *PTEN*. *PTEN* is a well-known prostate cancer-associated gene [153], which suggests that mutations in *TPTE* is highly likely to contribute to the development of PrCa. Although few studies suggesting the role of *TPTE* in cancer development, its absence in healthy tissues, homolog to *PTEN* and specific expression in testis [154] make it an attractive candidate for PrCa biomarker.

*KIR2DS3* is a member of killer-cell immunoglobulin receptor (KIR) family, which plays a crucial role in immune response. Various studies reflect divergent effects of KIR genes in cancers [155], thus their functions to tumorigenesis remain largely unknown. The remaining candidates in the list, including *PCDHA1* and *TCAF2*, have not been reported to be involved in any cancer development process, to the best of our knowledge. *PCDHGA2* can regulate Wnt signalling and growth properties in normal and cancer cells [156].

The nonfunctional pseudogene *CROCCP2* is possibly a false positive as it is barely reported to involve in biological activities. However, the other genes in Table 4.3 are promising cancer-associated genes. For some of them, their associations with PrCa remain largely unknown, but can be further explored.
5.4 SCORING FUNCTIONS

According to the rank sum of FATHMM and PROVEAN scores, the top 1% (54) pathogenic genes overlapped 9 LPC genes. When the same number of top genes selected according to occurrence frequency, only 4 LPC genes were included. The modest performance of these methods independently prompts us to develop some scoring functions that integrate all these prediction results. The $P_t(g)$ incorporates rank score of FATHMM and PROVEAN and the difference of subject counts between tumor and normal samples.

Noting that the length of CDS may affect the number of nsGSVs detected on the gene, a normalization factor involving the length of CDS was added to $P_t(g)$. Factors including CDS length itself, log and square root of CDS length were tried out and log transformation outperformed the others, thus was selected to normalize $P_t(g)$ and form the function $P_l(g)$. As the two scoring functions showed equivalent performance in detecting PrCa genes, both were used in selecting top pathogenic genes.

The pathogenic genes selected by the combination of two scoring functions cover 12 LPC genes, which validates its effectiveness in computationally selecting potentially pathogenic genes from a large set of genes. TP53 and SPOP are top two pathogenic genes identified by $P_t(g)$ and $P_l(g)$. These two genes have been extensively recognized as frequently mutated genes in PrCa [102,123], and TP53 is a recognized tumor suppressor gene across various cancer types [120,157].

5.5 BIOINFORMATICS ANALYSIS

GO term analysis of top 1% pathogenic genes revealed that these genes were largely enriched in the process related to the formation of cardiac muscle tissue, heart morphogenesis and contraction, and gated channel activities. There are increasing evidence suggesting that ion channels and calcium channels contribute to many cell behaviors that could be important to tumor
progression and metastasis including migration, invasion, adhesion, etc [158, 159]. Heart contraction is a process initiated by activation of voltage-dependent Ca\(^{2+}\) channels [160] and abnormal functions like heart failure with reduced ejection fraction has a higher likelihood of cancer development [161]. Thus, mutations in these genes may cause dysregulation of cell-cell adhesion and cardiac muscle abnormality which initiate tumor cell development and migration.

Enriched pathways were mainly regarding to cancers, cardiomyopathy and signaling pathways. Proteoglycans, as a functional constituents of extracellular matrix binding to various growth factors and adhesive factors, have been associated with cancer pathogenesis [162]. Proteins in calcium signaling pathway are important in regulating cell cycles such as proliferation, gene transcription and cell death by controlling Ca\(^{2+}\) pumps, Ca\(^{2+}\) channels, etc. [163]. Shapero et al. [164] in 2001 identified a prostate-specific gene which is homologous to the transient receptor potential family of Ca\(^{2+}\) channel proteins. Increased focal-adhesion kinase expression and activity have been observed in various cancer types. It was also reported to mediate tumorigenic activity, which is probably linked to its ability to control cell adhesion and migration, as well as to influence cell-survival pathways [165].

By constructing PPI networks, connections between pathogenic proteins were visualized. Screening of such genes and clusters from the network could help to confirm the essential genes that are involved in the pathogenesis of PrCa. Among these, \(TTN\) is strongly correlated with an emerging biomarker based on tumor mutational burden [166, 167]. While most mutations on this gene are likely to be passenger mutations, such abundant mutations in tumor tissues still encourage one to speculate its role in cancer development. \(TP53\) and \(ATM\) are well-established driver genes for PrCa while \(TRRAP\) is the driver for other cancers [120]. Among the 15 non-LPC pathogenic genes with high degrees of connectivity (≥ 4) with LPC genes in Table 4.5, \(CHD4\), \(RYBP\), \(NKX3-\)
1 and CSMD3 have either direct or indirect suppressive roles in PrCa [108,126,129,137]. Most others are involved in the progression of other cancers as shown in the References column of Table 4.5. How these genes affect cancer development is not yet understood, making them intriguing candidates for further investigation.
Chapter 6 Conclusions and future work

In this research, we have proposed a novel method to integrate functional effect predictions with frequency-based analysis of nsGSV and select pathogenic genes according to the cumulative scores of GSVs in the gene. After enrichment analyses were performed on the selected top pathogenic genes, their associations with known PrCa genes were revealed by a PPI network. This workflow is incorporated into a new version of the web-based OncoMiner pipeline, which can be publicly accessible by the scientific community.

When our method was applied to PrCa data, we not only identified a good number of genes that are well-known to drive PrCa but also uncovered several novel genes with strong association with PrCa based on a combined assessment of their general functional effects by FATHMM and PROVEAN, observed occurrence frequencies in this dataset, and protein interactions with well-known PrCa-related genes. While further studies on the specific functions of these genes and how they participate in cancer formation and development are necessary in order to provide solid evidence of their functional roles, they could be promising candidates to serve as biomarkers for the diagnosis of PrCa.

It should be noted that the prediction results from FATHMM and PROVEAN play important roles in the selection of high-scoring genes. As these types of functional effect prediction tools increase in availability and quality, they can be incorporated to our current scoring scheme to further improve the reliability of our proposed computational approach and reduce the work of laboratory validation. Moreover, we expect that our method would not only work on single nucleotide substitution GSVs, but can be easily extended to handle other small-scale mutations including indels. These could be good follow-up implementation projects that will make useful contributions and enhance the flexibility and versatility of the OncoMiner pipeline.
References


141. Amdahl G (1967) Validity of single-processor approach to achieving large-scale computing capability, proceedings of AFIPS conference, reston. VA.


Appendix

A File conversion and variant annotation

```python
import vcf
import csv
import time
import timeit
import gc

def writecsv(inputvcf):
    chr_start = time.time()
    vcf_reader = vcf.Reader(open(inputvcf, 'r'))
    outfile_name=[]
    col_name=
    index=[
        'var_index',
        'chrom',
        'left',
        'right',
        'ref_seq',
        'var_seq1',
        'var_seq2',
        'count1',
        'count2',
        'var_score',
        'gene_name',
        'where_in_transcript'
    ]
    index=[
        'ID',
        'genotype',
    ]
    chrom=[
    ]
    right=[
    ]
    ref=[
    ]
    alt=[
    ]
    alt2=[
    ]
    tags=[
    ]
    names=locals()
    sample_names=[
    ]
    sample_temps=[
    ]
    base_pool=[
        [ 'A' ],
        [ 'C' ],
        [ 'G' ],
        [ 'T' ]
    ]

    record=next(vcf_reader)
    # dynamically names output file for each sample
    for i in range(0,len(record.samples)):
        names['%s' %i].join([char for char in record.samples[i].sample if char.isalnum()])
        names['%s' %i].join([char for char in record.samples[i].sample.lower() if char.isalnum()])
        names['count1%s' %i]=[
        ]
        names['count2%s' %i]=[
        ]
        #names['GT%s' %i]=[]
        #names['GT%s' %i]=[]
        #names['GT%s' %i]=[
        ]
        new
        sample_temps.append(''.join([char for char in record.samples[i].sample if char.isalnum()]))
        sample_names.append(''.join([char for char in record.samples[i].sample.lower() if char.isalnum()]))
        vcf_reader = vcf.Reader(open(inputvcf, 'r'))
        for record in vcf_reader:
            chrom.append(record.CHROM.split())
            right.append(record.POS)
            left.append(str(record.POS).split())
            ref.append(record.REF.split())
            alt.append(record.ALT)
```

tags.append(record.FORMAT.split(':'))
for j in range(0, len(record.samples)):
    eval(sample_temps[j]).append(record.samples[j])
for m in range(0, len(tags)):
    for k in range(0, len(record.samples)):
        names['temp%s' %k]=[]
    for n in range(0, len(tags[0])):
        if tags[0][n] not in tags[m][n]:
            for j in range(0, len(record.samples)):
                eval('temp'+str(j)).append(None)
                tags[m].insert(n, tags[0][n])
        else:
            for j in range(0, len(record.samples)):
                eval('temp'+str(j)).append(eval(sample_temps[j])[m][tags[0][n]])
def get_index(input_list):
    AD=-1
    BCOUNT=-1
    for j in range(0, len(input_list)):
        if (input_list[j]=='AD'):
            AD=j
            return AD
        elif (input_list[j]=='BCOUNT'):
            BCOUNT=j
            return BCOUNT
        else:
            continue
    return
index_count=get_index(tags[0])
if (len(chrom)==len(left)==len(ref)==len(alt)):
    for j in range(0, len(record.samples)):
        names['list0%s' %j]=[[i]*len(chrom)
        names['temp%s' %j]=[[i]*len(chrom)
        index=[num for num in range(1, (len(chrom)+1))]
        score=[[28]]*len(chrom)
        alt2=['']*len(chrom)
    for i in range(0, len(chrom)):
        if(len(alt[i])>1):
            alt2[i]=[str(alt[i][1])]
            alt[i]=[str(alt[i][0])]
        else:
            if(ref[i][0]<str(alt[i][0])):
                alt2[i]=[str(alt[i][0])]
                alt[i]=ref[i]
            else:
                alt2[i]=ref[i]
                alt[i]=[str(alt[i][0])]
    index[i]=[str(index[i])]
right[i]=str(right[i]+len(ref[i]))

#for j in range(0,len(record.samples)):
This is new
    
    #for j in range(0,len(record.samples)):
    
    eval('GT'+str(j)).append(["",".join(eval(sample_names[j])[i][0].split('/'))])

if('BCOUNT' in tags[0]):
pos1=base_pool.index(alt[i])
pos2=base_pool.index(alt2[i])
    for k in range(0,len(record.samples)):
        eval('count1'+str(k)).append([eval(sample_names[k])[i][index_count][pos1]])
        eval('count2'+str(k)).append([eval(sample_names[k])[i][index_count][pos2]])

elif('AD' in tags[0]):
    if(alt[i]==ref[i]):
        for k in range(0,len(record.samples)):
            eval('count1'+str(k)).append([eval(sample_names[k])[i][index_count][0]])
            eval('count2'+str(k)).append([eval(sample_names[k])[i][index_count][1]])
    elif(alt2[i]==ref[i]):
        for k in range(0,len(record.samples)):
            eval('count1'+str(k)).append([eval(sample_names[k])[i][index_count][1]])
            eval('count2'+str(k)).append([eval(sample_names[k])[i][index_count][0]])
else:
    for k in range(0,len(record.samples)):
        eval('count1'+str(k)).append([eval(sample_names[k])[i][index_count][0]])
        eval('count2'+str(k)).append([eval(sample_names[k])[i][index_count][0:len(eval(sample_names[k])[i][index_count])]])
        else:
            break
        for j in range(0,len(record.samples)):
            eval('list0'+str(j))[i]= index[i]+chrom[i]+left[i]+ right[i]+ ref[i]+
            eval('temp'+str(j))=eval('list0'+str(j))[i]+eval('count1'+str(j))[i]+eval('count2'+str(j))[i]+score[i]#
            #eval('temp'+str(j))[i]=eval('list0'+str(j))[i]
            eval('temp'+str(j))[i]= index[i]+chrom[i]+
            left[i]+ right[i]+ ref[i]+
            eval('count1'+str(j))[i]+eval('count2'+str(j))[i]+score[i]

            for p in range(0,len(record.samples)):
                for ele in eval('temp'+str(p)):
if (type(ele[8])==int):
    if ((ele[7]+ele[8])<=5 or ele[7]/(ele[7]+ele[8])<=0.05 or ele[8]/(ele[7]+ele[8])<=0.05):
        eval('list0'+str(p)).remove(ele)
    else:
        pos=eval('list0'+str(p)).index(ele)
        eval('list0'+str(p)).insert(pos+1,ele)
if (ele[4]<ele[5]):
    eval('list0'+str(p))[pos][8]=ele[8][0]
    eval('list0'+str(p))[pos][5]=ele[4]
    eval('list0'+str(p))[pos][6]=ele[5]
else:
    eval('list0'+str(p))[pos][7]=ele[8][0]
    eval('list0'+str(p))[pos][5]=ele[4]
    eval('list0'+str(p))[pos][6]=ele[4]
    eval('list0'+str(p))[pos][8]=ele[7]
if (ele[4]<ele[6]):
    eval('list0'+str(p))[pos+1][8]=ele[8][1]
    eval('list0'+str(p))[pos+1][5]=ele[4]
    eval('list0'+str(p))[pos+1][6]=ele[6]
else:
    eval('list0'+str(p))[pos+1][7]=ele[8][1]
    eval('list0'+str(p))[pos+1][5]=ele[6]
    eval('list0'+str(p))[pos+1][6]=ele[4]
    eval('list0'+str(p))[pos+1][8]=ele[7]
if ((eval('list0'+str(p))[pos+1][7]+eval('list0'+str(p))[pos+1][8])<=5 or eval('list0'+str(p))[pos+1][7]/(eval('list0'+str(p))[pos+1][7]+eval('list0'+str(p))[pos+1][8])<=0.05 or eval('list0'+str(p))[pos+1][8]/(eval('list0'+str(p))[pos+1][7]+eval('list0'+str(p))[pos+1][8])<=0.05):
    eval('list0'+str(p)).pop(pos+1)
if ((eval('list0'+str(p))[pos][7]+eval('list0'+str(p))[pos][8])<=5 or eval('list0'+str(p))[pos][7]/(eval('list0'+str(p))[pos][7]+eval('list0'+str(p))[pos][8])<=0.05 or eval('list0'+str(p))[pos][8]/(eval('list0'+str(p))[pos][7]+eval('list0'+str(p))[pos][8])<=0.05):
    eval('list0'+str(p)).pop(pos)
    index1=[num for num in range(1,(len(eval('list0'+str(p)))+1))]
    for q in range(len(eval('list0'+str(p)))):
        eval('list0'+str(p))[q][0]=index1[q]
    for h in range(0,len(record.samples)):
        names['file1%s' %h]=open(sample_names[h]+'.csv','a',newline='')
        outfile_name.append(sample_names[h]+'.csv')
        with eval('file1'+str(h)):
            writer=csv.writer(eval('file1'+str(h)))
            writer.writerow(col_name)
            writer.writerows(eval('list0'+str(h)))
for h in range(0,len(record.samples)):
from multiprocessing import Pool, Manager
import os
from math import fabs
import sys
from miscVar import AADict
import time
import csv
import gc

MAX_DIST = 5000

# Function for processing vcf, returns dict

def vcf_func(vcf_file, gtf, genome):
    chr_start = time.time()
    vcf_file = open('%s' % (vcffile), 'r')
    vcf = vcf_file.readlines()
    vcf_file.close()
    gtf_file = gtf
    f = open(gtf_file, 'r')
    flat = f.readlines()
    f.close()
    seqfile = open(genome, 'r')
    next(seqfile)
    sequence = seqfile.read().replace('
', '')
    seqfile.close()

    dictionary = {}
    position = 0
    chrom = ''
    # genestart = 0
    # genestop = 0
    start_time = time.time()
    for l in range(0, len(vcf)):
        if vcf[l].startswith('chr'):
            line = vcf[l].strip()
            vl = line.split()  # Split on all white space
            chrom = vl[0]
            position = int(vl[1])
ref=vl[3]
alt=vl[4]
gtfCounter = -1  #point to each line in gtf file
genestart = 0
genestop = 0

if position not in dictionary:
    boo = 0
    where = []
genename = []
change_type=[]
aachange=[]
counter=[]
strand_gene=[]

while boo == 0 and gtfCounter < len(flat):
    if position > genestart and position <= genestop and
gtfCounter < (len(flat)-1):  #GSV within gene
counter.append(gtfCounter)
strand_gene.append(flat[gtfCounter].split()[3])
    #if (gtfCounter < (len(flat)-1)):
        gtfCounter += 1
pgstop = genestop
genestart = int(flat[gtfCounter].split()[4])

#transcript start

#transcript end

genestop = int(flat[gtfCounter].split()[5])

#Moves to the next gene

gtfCounter += 1
pgstop = genestop #prev gene stop

#transcript start

genestart = int(flat[gtfCounter].split()[4])

#transcript end

elif position > genestop and gtfCounter < (len(flat)-1):

    #transcript start

genestop = int(flat[gtfCounter].split()[5])

    #transcript end

elif position < genestart and position > pgstop:
    boo = 1
else:
    boo = 1

if len(strand_gene)==0:
    strand_gene.append('')
strand = flat[gtfCounter].split()[3]  #ref strand direction

run_time = time.time() - start_time

if len(counter)==0:
    if gtfCounter == len(flat)-1 and position > genestop:
        if (position - genestop) > MAX_DIST:  #if variant position is far from the end of transcript
            genename.append('NoName')
            where.append('Not close to gene')
            change_type.append('')
aachange.append('')

        else:
            genename.append(flat[gtfCounter].split()[0])
    #else whether close to 3' or 5'
    if strand == '-':
        where.append('5\' untranscribed')
change_type.append('')
aachange.append('')

else:
    where.append('3\' untranscribed')
    change_type.append('')
    aachange.append('')

elif position < genestart and position > pgstop:
    # check if closer to start or end of the one before or current
    diff1 = fabs(position - genestart)  # current
    diff2 = fabs(position - pgstop)     # previous

    if diff1 <= diff2 and diff1 <= MAX_DIST:
        genename.append(flat[gtfCounter].split()[0])
        if strand == '-':
            where.append('3\' untranscribed')
            change_type.append('')
            aachange.append('')
        else:
            where.append('5\' untranscribed')
            change_type.append('')
            aachange.append('')
    
elif diff2 <= MAX_DIST:  # if closer to prev gene and within cutoff distance
        genename.append(flat[gtfCounter - 1].split()[0])
        strand = flat[gtfCounter - 1].split()[3]
        if strand == '-':
            where.append('5\' untranscribed')
            change_type.append('')
            aachange.append('')
        else:
            where.append('3\' untranscribed')
            change_type.append('')
            aachange.append('')
    
else:
    genename.append('NoName')
    where.append('Not close to gene')
    change_type.append('')
    aachange.append('')

else:
    continue

else:
    for i in range(len(counter)):
        genename.append(flat[counter[i]].split()[0])
        # transcript start; check
        tr_start = int(flat[counter[i]].split()[4])
        # transcript end; check
        tr_end = int(flat[counter[i]].split()[5])
        cd_start = int(flat[counter[i]].split()[6])  # CDS start
        cd_end = int(flat[counter[i]].split()[7])    # CDS end
        ex_starts = []
        ex_ends = []
        x = 0
        tmp = 'Unknown'
        ex_starts = [int(i) for i in flat[counter[i]].split()[-2].split(',')] [0:-1]]
ex_ends = [int(i) for i in flat[counter[i]].split()[-1].split(',')][0:-1]]

while tmp == 'Unknown' and x < len(ex_starts):
    if position > ex_starts[x] and position <= ex_ends[x]:  # if position in exon
        tmp = 'Exon'
    elif position <= ex_starts[x]:
        tmp = 'Intron'
    else:
        x+=1
if tmp == 'Intron' or tmp == 'Unknown':
    where.append('Intron')
    change_type.append('')
    aachange.append('')
    # see if position is in translated region:
else:
    tmp == 'Exon':
        if position > cd_start and position <= cd_end:
            tmp = 'Translated'
        else:
            tmp = 'Not translated'
    if tmp == 'Not translated':
        if cd_start == cd_end:  # ncRNAs
            where.append('Not translated, ncRNA')
            change_type.append('')
            aachange.append('')
        else:
            # utrs
            if position <= cd_start:
                if strand_gene[i] == '-':
                    where.append('Not translated, 3\ utr')
                    change_type.append('')
                    aachange.append('')
                else:
                    where.append('Not translated, 5\ utr')
                    change_type.append('')
                    aachange.append('')
            else:
                if strand_gene[i] == '-':
                    where.append('Not translated, 5\ utr')
                    change_type.append('')
                    aachange.append('')
                else:
                    where.append('Not translated, 3\ utr')
                    change_type.append('')
                    aachange.append('')
    elif tmp == 'Translated':
        where.append('CDS')
        ref_codon = ''
        var_codon = ''
        aalength = 0
        m = 0
toRev=0
#m = (position - cd_start) % 3
if strand_gene[i]=="+":
ex=0
#pre_ex=0
    while ex != (len(ex_ends)-1) and
position > ex_ends[ex]:
        if cd_start > ex_ends[ex]:
ex+=1
        elif cd_start > ex_starts[ex] and
cd_start < ex_ends[ex] and position > ex_ends[ex]:
            #pre_ex=ex
            m = ((ex_ends[ex] -
cd_start )+m)%3
        aalength+=(ex_ends[ex] -
cd_start)
ex+=1
    else:
        # pre_ex=ex
        m = ((ex_ends[ex]-
ex_starts[ex])+m)%3
        aalength+=(ex_ends[ex] -
ex_starts[ex])
ex+=1
    #print("ex is %d" %ex)
if cd_start > ex_starts[ex]:
m=(position-cd_start -1+m )%3
aalength+=(position-cd_start)
else:
m=(position-ex_starts[ex] -1+m)%3
aalength+=(position-ex_starts[ex])
if aalength%3==0:
aapos=aalength//3
else:
aapos=aalength//3 + 1

if len(ref)==len(alt)and
position<len(sequence)-2:
    if m==0 :
        if len(ref)==1:
ref_codon=ref+sequence[position]+sequence[position+1]
var_codon=alt+sequence[position]+sequence[position+1]
elif len(ref)==2:
    ref_codon=ref+sequence[position]
    var_codon=alt+sequence[position]
else:
    ref_codon=ref
    var_codon=alt
elif m==1:
    if len(ref)==1:
      ref_codon = sequence[position-2]+ref+ sequence[position]
var_codon = sequence[position - 2]+alt+sequence[position]

eif len(ref)==2:
    ref_codon = sequence[position-
    2]+ref
    var_codon = sequence[position-
    2]+alt

else:
    ref_codon = sequence[position-
    2]+ref[0:2]
    var_codon = sequence[position-
    2]+alt[0:2]

else:
    if len(ref)==1:
        ref_codon = sequence[position-
        3]+sequence[position-2]+ref
        var_codon = sequence[position-
        3]+sequence[position-2]+alt

eif len(ref)==2:
    ref_codon = 
    ref[1]+sequence[position]+sequence[position+1]
    var_codon = 
    alt[1]+sequence[position]+sequence[position+1]

else:
    ref_codon = 
    ref[1:]+sequence[position]
    var_codon = 
    alt[1:]+sequence[position]

else:
    ex = len(ex_starts)-1
    while ex != 0 and position < 
    int(ex_starts[ex]):
        if cd_end < ex_starts[ex]:
            ex-=1
        elif cd_end < ex_ends[ex] and cd_end > 
        ex_starts[ex] and position < ex_starts[ex]:
            m = ((cd_end - 
            ex_starts[ex])+m)%3
            aalength=(cd_end-ex_starts[ex])
            ex-=1
        else:
            m = ((ex_ends[ex]-
            ex_starts[ex])+m)%3
            aalength+=(ex_ends[ex]-
            ex_starts[ex])
            ex-=1
        if cd_end > ex_ends[ex]:
            m=(ex_ends[ex] - position +m)%3
            aalength+=(ex_ends[ex]-position)
        else:
            m=(cd_end - position + m)%3
            aalength+=(cd_end-position)
        if aalength%3==0:
            aapos=aalength//3
        else:
aapos=aalength//3 + 1

toRev=1
if m==0 :
    if len(ref)==1:
        ref_codon=ref+sequence[position-2]+sequence[position-3]
        var_codon=alt+sequence[position-2]+sequence[position-3]
    elif len(ref)==2:
        ref_codon=ref+sequence[position-2]
        var_codon=alt+sequence[position-2]
else:
    ref_codon=ref
    var_codon=alt
elif m==1:
    if len(ref)==1:
        ref_codon = sequence[position]+ref+sequence[position-2]
        var_codon = sequence[position]+alt+sequence[position-2]
    elif len(ref)==1:
        ref_codon = sequence[position]+ref
        var_codon = sequence[position]+alt
    else:
        ref_codon = sequence[position]+ref[0:2]
        var_codon = sequence[position]+alt[0:2]
else:
    if len(ref)==1:
        ref_codon = sequence[position+1]+sequence[position]+ref
        var_codon = sequence[position+1]+sequence[position]+alt
    elif len(ref)==1:
        ref_codon=ref[1]+sequence[position-2]+sequence[position-3]
        var_codon=alt[1]+sequence[position-2]+sequence[position-3]
    else:
        ref_codon=ref[1:]+sequence[position-2]
        var_codon=alt[1:]+sequence[position-2]

    ref_codon=ref_codon.upper()
    var_codon=var_codon.upper()

if toRev==1:
    intab = "ATCG"
    outtab = "TAGC"
    trantab = str.maketrans(intab, outtab)
var_codon = var_codon.translate(trantab)
ref_codon = ref_codon.translate(trantab)

var_codon = var_codon[0:3]
ref_codon = ref_codon[0:3]
ref_aa = ''
var_aa = ''
if var_codon in AADict:
    var_aa = AADict[var_codon]  #translate
if ref_codon in AADict:
    ref_aa = AADict[ref_codon]
aachange.append(ref_aa+str(aapos)+var_aa)
if var_aa == ref_aa:
    change_type.append('Synonymous')
else:
    change_type.append('Non-synonymous')
else:
    where.append('Ignore')
    change_type.append('')
else:
    where.append('Ignore')
    change_type.append('')
dictionary[position]=[chrom,genename,where,change_type,aachange]  #include strand_gene for SIFT format
    chr_end = time.time() - chr_start
delete vcf
    delete seqfile
    gc.collect()
return dictionary

AADict = {
    "TTT": "F", "TTC": "F", "TTA": "L", "TTG": "L",
    "TCT": "S", "TCC": "S", "TCA": "S", "TCG": "S",
    "TAT": "Y", "TAC": "Y", "TAA": "*", "TAG": "*",
    "TGT": "C", "TGC": "C", "TGA": "*", "TGG": "W",
    "CTT": "L", "CTC": "L", "CTA": "L", "CTG": "L",
    "CCT": "P", "CCC": "P", "CCA": "P", "CCG": "P",
    "CAT": "H", "CAC": "H", "CAA": "Q", "CAG": "Q",
    "CGT": "R", "CGC": "R", "CGA": "R", "CGG": "R",
    "ATT": "I", "ATC": "I", "ATA": "I", "ATG": "M",
    "ACT": "T", "ACC": "T", "ACA": "T", "ACG": "T",
    "AAT": "N", "AAC": "N", "AAA": "K", "AAC": "K",
    "AGT": "S", "AGC": "S", "AGA": "R", "AGG": "R",
    "GTT": "V", "GTC": "V", "GTA": "V", "GTC": "V",
    "GCT": "A", "GCC": "A", "GCA": "A", "GCG": "A",
    "GAT": "D", "GAC": "D", "GAA": "E", "GAG": "E",
    "GGT": "G", "GGC": "G", "GGA": "G", "GGG": "G",}

from parse_VCF import vcf_func
from VCF2CSV import writecsv
#from functools import partial
import csv
import vcf

86
import gc
import time
# import sys
import os
import xlwt
from xlutils.copy import copy
from xlrd import open_workbook
import multiprocessing
import argparse
# import resource

# Function combines all information and writes an OMI file
def writeOMI(infile, outfile, newfile):
    start = time.time()
    f1 = open(outfile, 'r')
    rows = csv.reader(f1)
    f2 = open(newfile, 'a', newline='')
    writer = csv.writer(f2)
    for row in rows:
        for key in infile:
            if (row[1] == infile[key][0] and row[2] == str(key)):
                row.append(infile[key][1])
                row.append(infile[key][2])
                row.append(infile[key][3])
                row.append(infile[key][4])
                writer.writerow(row)
                break
    f1.close()
    f2.close()
    end = time.time() - start
    del rows
    gc.collect()

# Function that splits VCF file by chromosome for parallel processing
def splitvcf(vcffile):
    start = time.time()
    for i in range(1, 23):
        vcf_reader = vcf.Reader(open('%s %vcffile', 'r'))
        outvcf = open('chr' + str(i) + '.vcf', 'w')
        my_out = vcf.Writer(outvcf, vcf_reader)
        for record in vcf_reader:
            if record.CHROM == 'chr' + str(i):
                my_out.write_record(record)
        outvcf.close()
    vcf_reader = vcf.Reader(open('%s %vcffile', 'r'))
    outvcf1 = open('chrX.vcf', 'w')
    outvcf2 = open('chrY.vcf', 'w')
    my_out1 = vcf.Writer(outvcf1, vcf_reader)
    my_out2 = vcf.Writer(outvcf2, vcf_reader)
for record in vcf_reader:
    if (record.CHROM== 'chrX'):
        my_out1.write_record(record)
    elif(record.CHROM== 'chrY'):
        my_out2.write_record(record)
    else:
        continue
outvcf1.close()
outvcf2.close()
end=time.time()-start
del vcf_reader
gc.collect()
print('Time for generating vcf files:'+str(round(end))+'s')
return

#Function that splits reference file by chromosome for parallel processing

def splitgtf(gtfdir):
    start=time.time()
    filegtf=open(gtfdir,'r')
    flat=filegtf.readlines()
    filegtf.close()
    resultx=[]
    resulty=[]
    for i in range(1,23):
        myout=open('chr'+str(i)+'_ref.txt','a')
        result=[]
        for line in flat:
            if (line.split()[2]== 'chr'+str(i)):
                result.append(line)
            else:
                continue
        result=sorted(result, key=lambda x:int(x.split()[4]))
        myout.writelines(result)
        myout.close()
    for line in flat:
        if (line.split()[2]== 'chrX'):
            resultx.append(line)
        elif (line.split()[2]== 'chrY'):
            resulty.append(line)
        else:
            continue
    myoutx=open('chrX_ref.txt','a')
    myouty=open('chrY_ref.txt','a')
    resultx.sort()
    resulty.sort()
    myoutx.writelines(resultx)
    myouty.writelines(resulty)
    myoutx.close()
    myouty.close()
end=time.time()
del flat
del result
del resultx
del resulty
gc.collect()
print('Time for generating gtf files:' +str(round(end-start))+'s')

#Function that counts total number of variants, chromosomes and variants per chromosome
def count_chrom(vcfdir):
    vcffile=open('%s' % (vcfdir), 'r')
    vcfall = vcffile.readlines()
    chrom=[]
    var_chr=[]
    var_total=0
    count=0
    chrpool=['chrX','chrY']
    for i in range(1,23):
        chrpool.insert(i-1,'chr'+str(i))
    for k in range(len(vcfall)):
        line = vcfall[k].strip()
        vl = line.split()
        if(vl[0] in chrpool):
            var_total+=1
            if(vl[0] not in chrom):
                chrom.append(vl[0])
                var_chr.append(count)
                count=1
            else:
                count+=1
        var_chr.append(count)

del var_chr[0]
info=[chrom,var_chr,var_total]
vcfile.close()
del vcfall
del chrpool
gc.collect()
return info

if __name__ == '__main__':
    #vcfdir = input("vcf directory:")
    #outdir = input("output directory:")
    #gtfdir = input("gtf file:")
    #cores = int(input("number of cores:"))
    parser = argparse.ArgumentParser(description='Process VCF files to generate OMI files')
    parser.add_argument('-i', '--input_dir', metavar="DIRNAME",
dest='vcf_dir', required=True, help='Required: Input directory for all VCF files.')
parser.add_argument('-o', '--out_dir', metavar="DIRNAME",
dest='outdir', required=True, help='Required: Name of output directory to
store result files.')
parser.add_argument('-f', '--ref_flat', metavar="REFFLAT",
dest='ref_flat', required=True, help='Required: Reference genomic field
used to identify gene region, including path.')
parser.add_argument('-c', '--num_cores', metavar="NUMCORES",
dest='cores', type=int, default=3)
args = parser.parse_args()

vcfdir = args.vcf_dir
outdir = args.outdir
cores = args.cores
gtfdir = args.ref_flat

vcfs=os.listdir(vcfdir)
vcfs=[value for value in vcfs if value.endswith('vcf')] #remove non-vcf
files under vcfdir

(refpath, gtfname) = os.path.split(gtfdir)

if(not os.path.exists('runtime.xls')):
    workbook=xlwt.Workbook()
    sheet1=workbook.add_sheet('sheet1',cell_overwrite_ok=True)
    colname=['filename','Size','# GSVs','# chroms','# of samples']
    for i in range(len(colname)):
        sheet1.write(0,i,colname[i])
    workbook.save('runtime.xls')

for file in vcfs:
    #memory usage at the beginning
    starting = time.time()
    time0=splitecf(vcfdir+'/'+file)
splitgtf(gtfdir)

    info=count_chrom(vcfdir+'/'+file)
    (filename,extension)=os.path.splitext(file)
    filesize=os.path.getsize(vcfdir+'/'+file)/(1024*1024)
    if ('chrM' in info[0]):
        info[0].remove('chrM')
    vcf_by_chrom=[b+'.vcf' for b in info[0]]
    time1=time.time()

    pool=multiprocessing.Pool(processes=cores,maxtasksperchild=1)
    pool_list1=[]
    pool_list1=[pool.apply_async(writecsv,(vcf_by_chrom[i],)) for i in range(len(info[0]))]
    result1=[c.get() for c in pool_list1]
    csv_name=result1[0]

    pool.close()
    pool.join()
    time2=time.time()-time1
print("Time for writing all csv: %.2f seconds" % (time2))

for i in range(0, len(csv_name)):  
    f1 = open(csv_name[i], 'r')  
    header = f1.readline().strip('
')  
    # header1.append('gene_name')  
    header1.append('change_type1')  
    header1.append('AAchange
')  
    header1 = ','.join(header1)  
    f2 = open(outdir + filename + '_' + csv_name[i], 'a')  
    f2.write(header1)  
    f1.close()  
    f2.close()

# cores = multiprocessing.cpu_count() - 1  
pool = multiprocessing.Pool(processes=cores, maxtasksperchild=1)

for i in range(len(info[0])):  
    result2.append(c.get() for c in pool_list)  
    time4 = time.time() - time3  
    print("Time for parsing whole vcf: %.2f seconds" % (time4))

for j in range(0, len(csv_name)):  
    pool_list = pool.apply_async(writeOMIC, (result2[i], csv_name[j], outdir + filename + '_' + csv_name[j],))

else:  
    result = result2[0].copy()  
    for j in range(1, len(result2)):  
        result.update(result2[j])

for k in range(len(csv_name)):  
    pool_list = pool.apply_async(writeOMIC, (result, csv_name[k], outdir + filename + '_' + csv_name[k],))

pool.close()  
pool.join()  

time6 = time.time() - time5  
print("Time for writing all OMI: %.3f records" % (time6))

chrpool = ['chrX', 'chrX']  
for i in range(1, 23):  
    chrpool.insert(i - 1, 'chr' + str(i))

for i in chrpool:  
    os.remove(i + '_ref.txt')

for k in chrpool:
os.remove(k+'.vcf')
for j in csv_name:
    os.remove(j)

#en_mem=resource.getusage(resource.RUSAGE_SELF).ru_maxrss/1024#maximum
memory usage during the process

times=time.time()-starting

######## delete intermediate variables to release memory
    del result1
    del result2
    del pool_list
    del pool_list1
    gc.collect()

######## record runtime information to excel file
    rexcel=open_workbook('runtime.xls')
    filelist=rexcel.sheet_by_name('sheet1').col_values(0)
    header=rexcel.sheet_by_name('sheet1').row_values(0)
    rows=rexcel.sheets()[0].nrows
    excel=copy(rexcel)
    table=excel.get_sheet(0)

    if (filename in filelist):
        file_index=filelist.index(filename)
        if (cores in header):
            core_index=header.index(cores)
            table.write(file_index,core_index,times)
            table.write(file_index,core_index+1,time0)
            table.write(file_index,core_index+2,time2)
            table.write(file_index,core_index+3,time4)
            table.write(file_index,core_index+4,time6)
        else:
            table.write(0,len(header),cores)
            table.write(0,len(header)+1,'generating vcf')
            table.write(0,len(header)+2,'copy csv')
            table.write(0,len(header)+3,'parsing')
            table.write(0,len(header)+4,'write OMI')
            table.write(file_index,len(header),times)
            table.write(file_index,len(header)+1,time0)
            table.write(file_index,len(header)+2,time2)
            table.write(file_index,len(header)+3,time4)
            table.write(file_index,len(header)+4,time6)

    else:
        new_row=[filename,filesize,info[2],len(info[0]),len(csv_name)]
        for i in range(len(new_row)):
            table.write(rows,i,new_row[i])

        if (cores in header):
            core_index=header.index(cores)
            table.write(rows,core_index,times)
            table.write(rows,core_index+1,time0)
            table.write(rows,core_index+2,time2)
            table.write(rows,core_index+3,time4)
            table.write(rows,core_index+4,time6)
else:
    table.write(0, len(header), cores)
    table.write(0, len(header) + 1, 'splitvcf')
    table.write(0, len(header) + 2, 'copy csv')
    table.write(0, len(header) + 3, 'parsing')
    table.write(0, len(header) + 4, 'write OMI')
    table.write(rows, len(header), times)
    table.write(rows, len(header) + 1, time0)
    table.write(rows, len(header) + 2, time2)
    table.write(rows, len(header) + 3, time4)
    table.write(rows, len(header) + 4, time6)

    excel.save('runtime.xls')

    # print out filename and total time used
    print('Time elapsed:' + str(round(times, 2)) + ' seconds')
    print('Number of cores used:%d' % (cores))
    print('VCF filename:%s' % (file))
    # print(st_mem)
    # print(en_mem)
import numpy as np
import os
import pandas as pd
import argparse

def subject_count(tumordir, normaldir, outdir):
    #tumordir="D:\Personal
    #materials\PhD\Oncominer\Data\PRODOMI\OMI_PROD"
    #normaldir="D:\Personal
    #materials\PhD\Oncominer\Data\PRODOMI\OMI_NORMAL"
    tumors=os.listdir(tumordir)
    normals=os.listdir(normaldir)
    tumor=[value for value in tumors if value.endswith('tumor.csv')]
    normal=[value for value in normals if value.endswith('normal.csv')]
    tumorname=[value.split("_")[0] for value in tumor]
    normalname=[value.split("_")[0] for value in normal]
    del tumors, normals, tumor, normal

    tumorvar=[]
    normalvar=[]
    commonvar=[]
    counttumor=[]
    countnormal=[]
    countcommon=[]
    numtum=0
    numnor=0
    numcom=0

    if(normalname==tumorname):
        disease_var=pd.DataFrame()
        normal_var=pd.DataFrame()
        common_var=pd.DataFrame()
        for name in tumorname:
            tf=pd.read_csv(tumordir+name+'_tumor.csv')
            nf=pd.read_csv(normaldir+name+'_normal.csv')
            norepeat_tf = tf.drop_duplicates(subset=['chrom',
                'left','right','ref_seq','var_seq1',
                'var_seq2'],
            keep='first')
            norepeat_nf = nf.drop_duplicates(subset=['chrom',
                'left','right','ref_seq','var_seq1',
                'var_seq2'],
            keep='first')
            tidx=[]
            nidx=[]
            for i in range(norepeat_tf.shape[0]):
                for j in range(norepeat_nf.shape[0]):
                    if (norepeat_tf.iloc[i,1:7].values.tolist() ==
                        norepeat_nf.iloc[j,1:7].values.tolist()):
                        tidx.append(i)
                        nidx.append(j)
                        break
            disease_var=norepeat_tf.iloc[tidx].append(norepeat_nf.iloc[nidx])
            counttumor.append(len(tidx))
            countnormal.append(len(nidx))
            countcommon.append(len(tidx) + len(nidx) - len(tidx) - len(nidx))
            numtum+=len(tidx)
            numnor+=len(nidx)
            numcom+=len(tidx) + len(nidx) - len(tidx) - len(nidx)
            del norepeat_tf,norepeat_nf,
tidex,nidx
disease_var.to_csv(outdir+"\tumor\"+name+"_tumor.csv")
        normal_var=norepeat_tf.append(norepeat_nf)
        countnormal.append(len(norepeat_nf))
        numnor+=len(norepeat_nf)
disease_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
        counttumor.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numtum+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
disease_var.to_csv(outdir+"\common\"+name+"_common.csv")
        countnormal.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numnor+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
        normal_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
    else:
        for name in tumorname:
            tf=pd.read_csv(tumordir+name+'_tumor.csv')
            norepeat_tf = tf.drop_duplicates(subset=['chrom',
                'left','right','ref_seq','var_seq1',
                'var_seq2'],
            keep='first')
            tidx=[]
            nidx=[]
            for i in range(norepeat_tf.shape[0]):
                for j in range(norepeat_nf.shape[0]):
                    if (norepeat_tf.iloc[i,1:7].values.tolist() ==
                        norepeat_nf.iloc[j,1:7].values.tolist()):
                        tidx.append(i)
                        nidx.append(j)
                        break
            disease_var=norepeat_tf.iloc[tidx].append(norepeat_nf.iloc[nidx])
            counttumor.append(len(tidx))
            countnormal.append(len(nidx))
            countcommon.append(len(tidx) + len(nidx) - len(tidx) - len(nidx))
            numtum+=len(tidx)
            numnor+=len(nidx)
            numcom+=len(tidx) + len(nidx) - len(tidx) - len(nidx)
            del norepeat_tf,norepeat_nf,
tidex,nidx
disease_var.to_csv(outdir+"\tumor\"+name+"_tumor.csv")
        normal_var=norepeat_tf.append(norepeat_nf)
        countnormal.append(len(norepeat_nf))
        numnor+=len(norepeat_nf)
disease_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
        counttumor.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numtum+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
disease_var.to_csv(outdir+"\common\"+name+"_common.csv")
        countnormal.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numnor+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
        normal_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
        for name in normalname:
            nf=pd.read_csv(normaldir+name+'_normal.csv')
            norepeat_nf = nf.drop_duplicates(subset=['chrom',
                'left','right','ref_seq','var_seq1',
                'var_seq2'],
            keep='first')
            nidx=[]
            for i in range(norepeat_nf.shape[0]):
                for j in range(norepeat_tf.shape[0]):
                    if (norepeat_tf.iloc[i,1:7].values.tolist() ==
                        norepeat_nf.iloc[j,1:7].values.tolist()):
                        nidx.append(j)
                        break
            disease_var=norepeat_tf.iloc[nidx].append(norepeat_nf.iloc[nidx])
            counttumor.append(len(nidx))
            countnormal.append(len(tidx))
            countcommon.append(len(tidx) + len(nidx) - len(tidx) - len(nidx))
            numtum+=len(nidx)
            numnor+=len(tidx)
            numcom+=len(tidx) + len(nidx) - len(tidx) - len(nidx)
            del norepeat_tf,norepeat_nf,
tidx,nidx
disease_var.to_csv(outdir+"\tumor\"+name+"_tumor.csv")
        tumorvar.append(counttumor)
        normalvar.append(countnormal)
        commonvar.append(countcommon)
        counttumor=[]
        countnormal=[]
        countcommon=[]
        del tumors, normals, tumor, normal
        for name in tumorname:
            tf=pd.read_csv(tumordir+name+'_tumor.csv')
            norepeat_tf = tf.drop_duplicates(subset=['chrom',
                'left','right','ref_seq','var_seq1',
                'var_seq2'],
            keep='first')
            tidx=[]
            nidx=[]
            for i in range(norepeat_tf.shape[0]):
                for j in range(norepeat_nf.shape[0]):
                    if (norepeat_tf.iloc[i,1:7].values.tolist() ==
                        norepeat_nf.iloc[j,1:7].values.tolist()):
                        tidx.append(i)
                        nidx.append(j)
                        break
            disease_var=norepeat_tf.iloc[tidx].append(norepeat_nf.iloc[nidx])
            counttumor.append(len(tidx))
            countnormal.append(len(nidx))
            countcommon.append(len(tidx) + len(nidx) - len(tidx) - len(nidx))
            numtum+=len(tidx)
            numnor+=len(nidx)
            numcom+=len(tidx) + len(nidx) - len(tidx) - len(nidx)
            del norepeat_tf,norepeat_nf,
tidex,nidx
disease_var.to_csv(outdir+"\tumor\"+name+"_tumor.csv")
        normal_var=norepeat_tf.append(norepeat_nf)
        countnormal.append(len(norepeat_nf))
        numnor+=len(norepeat_nf)
disease_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
        counttumor.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numtum+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
disease_var.to_csv(outdir+"\common\"+name+"_common.csv")
        countnormal.append(len(norepeat_tf))
        countcommon.append(len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf))
        numnor+=len(norepeat_tf)
        numcom+=len(norepeat_tf) + len(norepeat_nf) - len(norepeat_tf)
        normal_var.to_csv(outdir+"\normal\"+name+"_normal.csv")
    return [counttumor, countnormal, countcommon]

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-t', '--tumordir', type=str, required=True)
    parser.add_argument('-n', '--normaldir', type=str, required=True)
    parser.add_argument('-o', '--outdir', type=str, required=True)
    args = parser.parse_args()
    print(subject_count(args.tumordir, args.normaldir, args.outdir))
tidx.append(i)
nidx.append(j)
break

a=pd.DataFrame()
norepeat_tf=a.append(norepeat_tf,ignore_index=True)
norepeat_nf=a.append(norepeat_nf,ignore_index=True)

common=a.append(norepeat_tf.loc[tidx],ignore_index=True)
tempcom=a.append(norepeat_tf.loc[tidx],ignore_index=True)
norepeat_tf=norepeat_tf.drop(tidx,axis=0)
norepeat_nf=norepeat_nf.drop(nidx,axis=0)
temptf=a.append(norepeat_tf,ignore_index=True)
tempnf=a.append(norepeat_nf,ignore_index=True)

for j in range(common.shape[0]):
    if (list(common.iloc[j,1:7]) in commonvar):
        countcommon[commonvar.index(list(common.iloc[j,1:7]))]+=1
        tempcom=tempcom.drop(j,axis=0)
    else:
        commonvar.append(list(common.iloc[j,1:7]))
        countcommon.append(1)
        numcom=numcom+tempcom.shape[0]
        common_var=common_var.append(tempcom,ignore_index=True)

for j in range(norepeat_tf.shape[0]):
    if (list(norepeat_tf.iloc[j,1:7]) in tumorvar):
        counttumor[tumorvar.index(list(norepeat_tf.iloc[j,1:7]))]+=1
        temptf=temptf.drop(j,axis=0)
    else:
        tumorvar.append(list(norepeat_tf.iloc[j,1:7]))
        counttumor.append(1)
        numtum=numtum+temptf.shape[0]
        disease_var=disease_var.append(temptf,ignore_index=True)

for j in range(norepeat_nf.shape[0]):
    if (list(norepeat_nf.iloc[j,1:7]) in normalvar):
        countnormal[normalvar.index(list(norepeat_nf.iloc[j,1:7]))]+=1
        tempnf=tempnf.drop(j,axis=0)
    else:
        normalvar.append(list(norepeat_nf.iloc[j,1:7]))
        countnormal.append(1)
        numnor=numnor+tempnf.shape[0]
        normal_var=normal_var.append(tempnf,ignore_index=True)

### keep count in each group

disease_var.insert(disease_var.shape[1],"count",counttumor)
normal_var.insert(normal_var.shape[1],"count",countnormal)
common_var.insert(common_var.shape[1], "count", countcommon)

transcript=disease_var.append(normal_var)

transcript=transcript.append(common_var)

transcript=transcript.drop_duplicates(subset=['chrom', 'left', 'right', 'ref_seq', 'var_seq1',
                                             'var_seq2'],
                                        keep='first')


t0n0=[0]*transcript.shape[0]
t0n1=[0]*transcript.shape[0]
t1n0=[0]*transcript.shape[0]
t1n1=[0]*transcript.shape[0]

for i in range(transcript.shape[0]):
    var=transcript.iloc[i, 0:7].tolist()

    for name in tumorname:
        tf=pd.read_csv(tumordir+name+"_tumor.csv")
        nf=pd.read_csv(normaldir+name+"_normal.csv")

        norepeat_tf = tf.drop_duplicates(subset=['chrom', 'left', 'right', 'ref_seq', 'var_seq1',
                                                   'var_seq2'],
                                           keep='first')
        norepeat_nf = nf.drop_duplicates(subset=['chrom', 'left', 'right', 'ref_seq', 'var_seq1',
                                                   'var_seq2'],
                                           keep='first')

        tfile=np.array(norepeat_tf.iloc[:, 1:7]).tolist()
        nfile=np.array(norepeat_nf.iloc[:, 1:7]).tolist()

        if var in tfile and var in nfile:
            t1n1[i] += 1
        elif var in tfile and var not in nfile:
            t1n0[i] += 1
        elif var not in tfile and var in nfile:
            t0n1[i] += 1
        else:
            t0n0[i] += 1

transcript.insert(transcript.shape[1], "subt1n1", t1n1)
transcript.insert(transcript.shape[1], "subt1n0", t1n0)
transcript.insert(transcript.shape[1], "subt0n1", t0n1)
transcript.insert(transcript.shape[1], "subt0n0", t0n0)

alt=[]

for k in range(transcript.shape[0]):
    if transcript.iloc[k, 4]==transcript.iloc[k, 5]:
        alt.append(transcript.iloc[k, 6])
    else:
        alt.append(transcript.iloc[k, 5])

transcript.insert(5, 'alt', alt)
transcript.drop(columns=['var_index', 'var_seq1', 'var_seq2'], inplace=True)

...
file_total_tumor = [0] * transcript.shape[0]
file_total_normal = [0] * transcript.shape[0]
for name in tumorname:
    tf = pd.read_csv(tumordir + name + "_tumor.csv")
    nf = pd.read_csv(normaldir + name + "_normal.csv")
    norepeat_tf = tf.drop_duplicates(subset=['chrom', 'left', 'right', 'ref_seq', 'var_seq1', 'var_seq2'], keep='first')
    norepeat_nf = nf.drop_duplicates(subset=['chrom', 'left', 'right', 'ref_seq', 'var_seq1', 'var_seq2'], keep='first')

    alt = []
    for k in range(norepeat_tf.shape[0]):
        if norepeat_tf.iloc[k, 4] == norepeat_tf.iloc[k, 5]:
            alt.append(norepeat_tf.iloc[k, 6])
        else:
            alt.append(norepeat_tf.iloc[k, 5])
    norepeat_tf.insert(5, 'alt', alt)
    norepeat_tf.drop(columns=['var_seq1', 'var_seq2'], inplace=True)

    alt = []
    for k in range(norepeat_nf.shape[0]):
        if norepeat_nf.iloc[k, 4] == norepeat_nf.iloc[k, 5]:
            alt.append(norepeat_nf.iloc[k, 6])
        else:
            alt.append(norepeat_nf.iloc[k, 5])
    norepeat_nf.insert(5, 'alt', alt)
    norepeat_nf.drop(columns=['var_seq1', 'var_seq2'], inplace=True)

per_file_tumor = [0] * transcript.shape[0]
per_file_normal = [0] * transcript.shape[0]
merge_tumor = pd.merge(transcript, norepeat_tf, on=['chrom', 'left', 'right', 'ref_seq', 'alt'], how='left')
include_tumor = merge_tumor[~merge_tumor['count1_y'].isnull()]
index_tumor = include_tumor._stat_axis.tolist()
for ele in index_tumor:
    per_file_tumor[ele] = 1
    file_total_tumor[ele] += 1

merge_normal = pd.merge(transcript, norepeat_nf, on=['chrom', 'left', 'right', 'ref_seq', 'alt'], how='left')
include_normal = merge_normal[~merge_normal['count1_y'].isnull()]
index_normal = include_normal._stat_axis.tolist()
for ele in index_normal:
    per_file_normal[ele] = 1
    file_total_normal[ele] += 1

transcript.insert(transcript.shape[1], name + '_tumor', per_file_tumor)
transcript.insert(transcript.shape[1], name+'_normal', per_file_normal)

# return transcript
transcript.to_csv(outdir+'sub_count_dist.csv', index=False)

if __name__ == '__main__':
    # tumordir=input("directory of tumor OMI file:")
    # normaldir=input("directory of normal OMI file:")
    # outdir=input("directory of output file:")
    parser = argparse.ArgumentParser(description='count subjects for each variant')
    parser.add_argument('-t', '--input_dir', metavar="TUMORDIR",
                        dest='tumor', required=True, help='Required: Directory to all tumor OMI files.')
    parser.add_argument('-o', '--out_dir', metavar="DIRNAME",
                        dest='outdir', required=True, help='Required: Name of output directory to store result file.')
    parser.add_argument('-n', '--input_dir', metavar="NORMALDIR",
                        dest='normal', required=True, help='Required: Directory to all normal OMI files.')
    args = parser.parse_args()

    tumordir = args.tumor
    normaldir = args.normal
    outdir = args.outdir
    subject_count(tumordir, normaldir, outdir)
 Functional predictions

```python
# -*- coding: utf-8 -*-
import os
import pandas as pd
import mechanicalsoup
from urllib.request import urlopen
from bs4 import BeautifulSoup
import time
import csv
import argparse

def fathmm(omifile, fatout, build='hg38', sleeptime=60):
    omi = pd.read_csv(omifile)
    directory, name = os.path.split(omifile)
    name = name.split('.')[0]
    del directory
    temp = omi[['left', 'ref_seq']]
    alt = []
    for i in range(omi.shape[0]):
        if omi.iloc[i, 5] == omi.iloc[i, 6]:
            alt.append(omi.iloc[i, 7])
        else:
            alt.append(omi.iloc[i, 6])
    chrom = omi['chrom'].apply(lambda x: x[3:])
    temp.insert(0, 'chrom', chrom)
    temp.insert(3, 'alt', alt)
    temp['left'] = temp['left'].apply(lambda x: str(x))
    temp.drop_duplicates()
    fathmm = temp.apply(lambda x: x['chrom'] + ',' + x['left'] + ',' + x['ref_seq'] + ',' + x['alt'], axis=1)
    del temp, i, alt, chrom, omi

    browser = mechanicalsoup.StatefulBrowser()
    browser.open("http://fathmm.biocompute.org.uk/fathmm-xf/")
    browser.get_url()
    browser.get_current_page()
    browser.select_form('form[name="myForm"]')
    browser['batch'] = '
'.join(fathmm.tolist())
    if build == 'hg38':
        browser['hg38'] = "hg38"
    response = browser.submit_selected()
    time.sleep(sleeptime)
    browser.get_url()
    temp = response.text.split("session=")
    session = temp[1].split('\"')[0]
    resulturl = "http://fathmm.biocompute.org.uk/fathmm-xf/cgi-bin/results.cgi?session=" + session
    #resultpage=urllib.request.urlopen(resulturl).read()
    resultpage = urlopen(resulturl).read()
    soup = BeautifulSoup(resultpage, 'html.parser')
```
table = soup.find('table', attrs={'class': 'table table-striped table-bordered'})
headings = [th.get_text() for th in table.find('tr').find_all('th')]
datasets = []
for row in table.find_all('tr')[1:]:
    dataset = [td.get_text() for td in row.find_all('td')]
datasets.append(dataset)
output = open(fatout + name + '_fathmm.csv', 'a', newline='')
writer = csv.writer(output)
writer.writerow(headings)
for ele in datasets:
    writer.writerow(ele)
output.close()
return

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='prepare for FATHMM submission and parse results')
    parser.add_argument('-i', '--input_file', metavar='FILENAME', dest='omi', required=True, help='Required: Input OMI file.')
    parser.add_argument('-o', '--output_dir', metavar='DIRNAME', dest='outdir', required=True, help='Required: Name of output directory to store results.')
    parser.add_argument('-r', '--reference_build', metavar='ref_build', dest='ref', default='hg38', help='Choose reference genome going to be used')
    parser.add_argument('-t', '--runtime', metavar='runtime', dest='time', type=int, default=60)
    args = parser.parse_args()
    omifile = args.omi
    outdir = args.outdir
    build = args.ref
    sleeptime = args.time
    fathmm(omifile, outdir, build, sleeptime)

# -*- coding: utf-8 -*-
import re
from miscVar import AADict
MAX_DIST = 5000
import bisect
import pandas as pd
import os
import argparse
def get_same_element_index(ob_list, word):
    return [i for (i, v) in enumerate(ob_list) if v == word]
intab = "ATCG"
outtab = "TAGC"
trantab = str.maketrans(intab, outtab)
def annotation(unify, build, flat):
    dictionary = {}
    position = 0
    chrom = ''
    prechrom = 0
    flatpath, flatname = os.path.split(flat)
    seqpath, seqname = os.path.split(build)

    for l in range(unify.shape[0]):  # line in vcf:
        chrom = unify['chrom'][l]
        position = unify['left'][l]
        ref = unify['ref_seq'][l]
        alt = unify['alt'][l]
        gtfCounter = -1
        genestart = 0
        genestop = 0
        boo = 0
        where = []
        genename = []
        change_type = []
        counter = []
        strand_gene = []
        aavar = []
        dnasequence = []
        codon_pos = []
        ref_trip = []
        var_trip = []
        if chrom != prechrom:
            f = open(flatpath + str(chrom) + flatname, 'r')
            flat = f.readlines()
            f.close()
            seqfile = open(seqpath + '_' + str(chrom) + '.fa', 'r')
            next(seqfile)
            sequence = seqfile.read().replace('
', '')
            seqfile.close()
            prechrom = chrom

        while boo == 0 and gtfCounter < len(flat):
            if position >= genestart and position <= genestop and gtfCounter < (len(flat) - 1):
                counter.append(gtfCounter)
                strand_gene.append(flat[gtfCounter].split()[3])

                gtfCounter += 1
                pgstop = genestop
                genestart = int(flat[gtfCounter].split()[4])  # transcript start
            else:
                genestop = int(flat[gtfCounter].split()[5])  # transcript end
                if position > genestop and gtfCounter < (len(flat) - 1):
                    gtfCounter += 1
                    pgstop = genestop  # prev gene stop
```python
genestart = int(flat[gtfCounter].split()[4])  # transcript start

genestop = int(flat[gtfCounter].split()[5])  # transcript end

eelif position < genestart and position > pgstop:
  boo = 1

else:
  boo = 1

if len(strand_gene)==0:
  strand_gene.append('')

for i in range(len(counter)):
  genename.append(flat[counter[i]].split()[0]+'_str(counter[i]))

  cd_start=int(flat[counter[i]].split()[6])  # CDS start

  cd_end=int(flat[counter[i]].split()[7])  # CDS end

  ex_starts = []

  ex_ends=[]

  dnaseq=''

  x=0

  tmp = 'Unknown'

  ex_starts = [int(i) for i in flat[counter[i]].split()[2].split(',')[0:-1]]

  ex_ends = [int(i) for i in flat[counter[i]].split()[1].split(',')[0:-1]]

  while tmp == 'Unknown' and x < len(ex_starts):
    if position > ex_starts[x] and position <= ex_ends[x]:  # if position in exon
      tmp = 'Exon'
      elif position < ex_starts[x]:
        tmp = 'Intron'
      else:
        x+=1

        if tmp == 'Intron' or tmp=='Unknown':
          where.append('Intron')

          change_type.append('')

          # see if position is in translated region:
          elif tmp == 'Exon':
            if position > cd_start and position <= cd_end:
              tmp = 'Translated'
            else:
              # not translated
              tmp = 'Not translated'

              if tmp == 'Not translated':
                if cd_start == cd_end:  # ncRNAs
                  where.append('Not translated, ncRNA')

                  change_type.append('')

                else:
                  # utrs
                  if position < cd_start:
                    if strand_gene[i] == '-':
                      where.append('Not translated, 3\' utr')

                      change_type.append('')
```

else:
    where.append('Not translated, 5\' utr')
    change_type.append('')
else:
    if strand_gene[i] == '-':
        where.append('Not translated, 5\' utr')
        change_type.append('')
else:
    where.append('Not translated, 3\' utr')
    change_type.append('')

elif tmp == 'Translated':
    where.append('CDS')
    ref_codon = ''
    var_codon = ''
    m = 0
    toRev = 0
    aapos = 0
    aaseq = ''
    
    #m = (position - cd_start) % 3
    if strand_gene[i] == '+':
        ex = 0
        while ex != (len(ex_ends) - 1) and position > ex_ends[ex]:
            if cd_start > ex_ends[ex]:
                ex += 1
            elif cd_start > ex_starts[ex] and cd_start < ex_ends[ex] and position > ex_ends[ex]:
                m = ((ex_ends[ex] - cd_start) + m) % 3
                aapos += (ex_ends[ex] - cd_start)
                ex += 1
            else:
                m = ((ex_ends[ex] - ex_starts[ex]) + m) % 3
                aapos += (ex_ends[ex] - ex_starts[ex])
                ex += 1
        if cd_start > ex_starts[ex]:
            m = (position - cd_start - 1 + m) % 3
            aapos += (position - cd_start)
        else:
            m = (position - ex_starts[ex] - 1 + m) % 3
            aapos += (position - ex_starts[ex])
    if aapos % 3 == 0:
        pos = aapos // 3
    else:
        pos = aapos % 3 + 1
        firstex = bisect.bisect(ex_starts, cd_start) - 1
        lastex = bisect.bisect(ex_ends, cd_end)
    if lastex >= len(ex_ends):
        lastex = len(ex_ends) - 1
    if ex_ends[firstex] > cd_start:
        dnaseq = dnaseq + sequence[cd_start:ex_ends[firstex]]
        firstex += 1
while firstex < lastex:

dnaseq=dnaseq+sequence[ex_starts[firstex]:ex_ends[firstex]]
    firstex+=1
    if cd_end > ex_starts[lastex]:

dnaseq=dnaseq+sequence[ex_starts[lastex]:cd_end]
    if m==0 :

ref_codon=ref+sequence[position]+sequence[position+1]

var_codon=alt+sequence[position]+sequence[position+1]
    codon_pos.append(1)

elif m==1:
    ref_codon = sequence[position-2]+ref+
sequence[position]
    var_codon = sequence[position-2]+alt+sequence[position]
    codon_pos.append(2)
else:
    ref_codon = sequence[position-3]+sequence[position-2]+ref
    var_codon = sequence[position-3]+sequence[position-2]+alt
    codon_pos.append(3)
else:

    ex = len(ex_starts)-1
    while ex != 0 and position < int(ex_starts[ex]):
        if cd_end < ex_starts[ex]:
            ex-=1
        elif cd_end < ex_ends[ex] and cd_end > ex_starts[ex] and position < ex_starts[ex]:
            m = ((cd_end - ex_starts[ex] )+m)%3
            aapos=(cd_end-ex_starts[ex])
            ex-=1
        else:
            m = ((ex_ends[ex]-ex_starts[ex] )+m)%3
            aapos+=(ex_ends[ex]-ex_starts[ex])
            ex-=1
    if cd_end > ex_ends[ex]:
        m=(ex_ends[ex] - position + m)%3
        aapos+=(ex_ends[ex]-position)
    else:
        m=(cd_end - position + m)%3
        aapos+=(cd_end-position)

    if aapos%3==0:
        pos=aapos//3
    else:
        pos=aapos//3 + 1

firstex=bisect.bisect(ex_starts, cd_start)-1
lastex=bisect.bisect(ex_ends,cd_end)
if lastex>=len(ex_ends):
    lastex=len(ex_ends)-1
if ex_ends[firstex]>cd_start:
    dnaseq = dnaseq+sequence[cd_start:ex_ends[firstex]]
    firstex+=1
while firstex < lastex:
    dnaseq=dnaseq+sequence[ex_starts[firstex]:ex_ends[firstex]]
    firstex+=1
if cd_end > ex_starts[lastex]:
    dnaseq=dnaseq+sequence[ex_starts[lastex]:cd_end]
    dnaseq=dnaseq[:::-1]
toRev=1
if m==0 :
codon_pos.append(1)
elif m==1:
    ref_codon = sequence[position]+ref+
    var_codon = sequence[position]+alt+sequence[position-2]
codon_pos.append(2)
else:
    ref_codon = sequence[position+1]+sequence[position]+ref
    var_codon = sequence[position+1]+sequence[position]+alt
codon_pos.append(3)
    ref_codon=ref_codon.upper()
    var_codon=var_codon.upper()
if toRev==1:
    intab = "ATCG"
    outtab = "TAGC"
    trantab = str.maketrans(intab, outtab)
    var_codon = var_codon.translate(trantab)
    ref_codon = ref_codon.translate(trantab)
var_codon = var_codon[0:3]
ref_codon = ref_codon[0:3]
ref_aa=''
var_aa=''
if var_codon in AADict:
    var_aa = AADict[var_codon] #translate
if ref_codon in AADict:
    ref_aa = AADict[ref_codon]
if var_aa == ref_aa:
    change_type.append('Synonymous')
else:
    change_type.append('Non-synonymous')
aavar.append(ref_aa+str(int(pos))+var_aa)
ref_trip.append(ref_codon)
var_trip.append(var_codon)
dna_sequence.append(dnaseq)
else:
    # To make sure to fill change type in on the ignored ones.
    where.append('Ignore')
    change_type.append('')
else:
    where.append('Ignore')
    change_type.append('')
dictionary[l]=[chrom,position,strand_gene,genename,where,change_type,dna_sequence]
    return dictionary

if __name__=='__main__':
    parser = argparse.ArgumentParser(description='prepare for PROVEAN submission')
    parser.add_argument('-i', '--input_file', metavar='DIRNAME',
                        dest='nonsyn', required=True, help='Required: Input OMI file with nonsynonymous variants.')
    parser.add_argument('-s', '--outseq_dir', metavar='DIRNAME',
                        dest='seqdir', required=True, help='Required: Name of output directory to store aa sequence.')
    parser.add_argument('-v', '--outvar_dir', metavar='DIRNAME',
                        dest='vardir', required=True, help='Required: Name of output directory to store aa variations.')
    parser.add_argument('-r', '--ref_sequence', metavar='ref_build',
                        dest='ref_seq', required=True, help='Required: Reference sequence file used to translate aa, including path. Sequence should be split by chromosome.')
    parser.add_argument('-f', '--ref_flat', metavar='ref_flat',
                        dest='ref_flat', required=True, help='Required: Reference genomic field used to identify gene region, including path. File should be split by chromosome.')
    args = parser.parse_args()
nonsynpath = args.nonsyn
aa_seqpath = args.semdir
aavarpath = args.var_dir
build = args.ref_build
flat = args.ref_flat
nonsyn=pd.read_csv(nonsynpath)
seqname=[]
result=annotation(nonsyn,build,flat)
for j in range(len(result)):
    nonsynidx=get_same_element_index(result[j][5], 'Non-synonymous')
    for n in range(len(nonsynidx)):
        varfile=open(aavarpath+result[j][3][nonsynidx[n]]+".var", "a")
varfile.write(nonsyn['AAchange'][j][nonsynidx[n]])
varfile.close()

if result[j][3][nonsynidx[n]] not in seqname:
    seqname.append(result[j][3][nonsynidx[n]])
if result[j][2][nonsynidx[n]]=='-':
    geneseq=result[j][6][n].upper().translate(trantab)
else:
    geneseq=result[j][6][n].upper()
aaseq=''
for i in range(0,len(geneseq),3):
    if len(geneseq)>=i+3 and AADict[geneseq[i:i+3]]!='*':
        aaseq=aaseq+AADict[geneseq[i:i+3]]
    else:
        break
aaseq=re.sub("(.{60})","\"\n",aaseq)
output=open(aaseqpath+result[j][3][nonsynidx[n]]+".fasta", "w")
output.write("> sp| " + result[j][3][nonsynidx[n]] + "\n")
output.write(aaseq)
output.close()
fw=open(aaseqpath+seqname[i]+".fasta","w")
fw.write("> sp|" + seqname[i] + "\n")
fw.write(sequence)
fw.close()
D Pathogenic scores calculation

```python
import pandas as pd
from collections import Counter
import math
import argparse

def get_same_element_index(ob_list, word):
    return [i for (i, v) in enumerate(ob_list) if v == word]

def collect_gene(var_df):
    singlegene=[]
    provean_min=[]
    fat_combine=[]
    for i in range(var_df.shape[0]):
        word=var_df['genename'][i].replace('\', '').replace('[', '').replace(']', '').split(',
        anno=var_df['where'][i].replace('\', '').replace('[', '').replace(']', '').split(',
        anno=list(filter(lambda x:x!='''Nottranslated' and
x!='''Nottranslated',anno))
        idx=get_same_element_index(anno,'CDS')
        word1=[]
        for ele in idx:
            word1.append(word[ele])
        singlegene.append(Counter(word1).most_common(1)[0][0])
        provean=var_df['provean score'][i].split(',')
        provean=list(map(lambda x:float(x),provean))
        provean_min.append(min(provean))
        if str(var_df['Coding Score'][i])==''nan'':
            fat_combine.append(float(var_df['Non-Coding Score'][i]))
        else:
            fat_combine.append(float(var_df['Coding Score'][i]))
    var_df.insert(var_df.shape[1],'single gene',singlegene)
    var_df.insert(var_df.shape[1],'provean min',provean_min)
    var_df.insert(var_df.shape[1],'fat combine',fat_combine)
    var_df.insert(var_df.shape[1],'provean rank',var_df['provean min'].rank())
    var_df.insert(var_df.shape[1],'fathmm rank',var_df['fat combine'].rank(ascending=False))
    var_df['rank sum']=var_df['fathmm rank']+var_df['provean rank']
    var_df['rank-score']=1-var_df['rank sum']/(2*var_df.shape[0])
    var_df=var_df.sort_values('rank sum',axis=0)
    return var_df

def select_cds(curr):
    maximum=0
    for i in range(curr.shape[0]):
        ex=0
        m=0
```

109
while ex != (len(curr.iloc[i,10])):
    if curr.iloc[i,6] > curr.iloc[i,10][ex]:
        ex+=1
    elif curr.iloc[i,6] > curr.iloc[i,9][ex] and curr.iloc[i,6] < curr.iloc[i,10][ex]:
        m = (curr.iloc[i,10][ex] - curr.iloc[i,6])+m
        ex+=1
    else:
        m = (curr.iloc[i,10][ex]-curr.iloc[i,9][ex])+m
        ex+=1
if m> maxium:
    maxium=m
return m

def pathogenic_score(var_df,gtf,out):
    gene_counts=var_df['single gene'].value_counts()
    patho_gene=gene_counts.reset_index()
    patho_gene.rename(columns={'index':'genename','single gene':'distinct pathogenic nsGSV'},inplace=True)
    cds_length=[]
    for i in range(patho_gene.shape[0]):
        curr=gtf[gtf[0]==patho_gene.iloc[i,0]]
        cds_length.append(select_cds(curr))
        patho_gene.insert(patho_gene.shape[1],'CDS length',cds_length)
    plg=[]
    ptg=[]
    for i in range(patho_gene.shape[0]):
        curr=var_df[var_df['single gene']==patho_gene.iloc[i,0]]
        plg.append(sum((curr['SubT1N0']-curr['SubT0N1'])*curr['rank-score']))
        #curr=curr[(curr['SubT1N0']+curr['SubT1N1'])*curr['SubT0N1']]+curr['SubT1N1']==0)
        ptg.append(sum((curr['SubT1N0']-curr['SubT0N1'])*curr['rank-score']))

    patho_gene.insert(patho_gene.shape[1],'Pt(g)',ptg)
    patho_gene.insert(patho_gene.shape[1],'Pl(g)',plg)
    rank=patho_gene['Pl(g)'].rank(ascending=False)
    patho_gene.to_csv(out,index=False)

if __name__=='__main__':
    parser = argparse.ArgumentParser(description='count subjects for each variant')
    parser.add_argument('-t', '--input_dir', metavar='nsGSV',
                        dest='nonsyn', required=True, help='Required: nsGSV files.')
    parser.add_argument('-f', '--ref_flat', metavar='REFFLAT',
                        dest='ref_flat', required=True, help='Required: Reference genomic field file.')
parser.add_argument('--o', '--out_dir', metavar="DIRNAME", dest='outfile', required=True, help='Required: Path and name of output file.')
    args = parser.parse_args()

    var_df = pd.read_csv(args.nonsyn)
    gtf = pd.read_table(args.ref_flat)
    outdir = args.outdir
    var_df = collect_gene(var_df)
    pathogenic_score(var_df, gtf, outdir)
Vita

Bofei Wang was born in a medium-sized city, Xiangyang, in midland of China. Being the only child in the family, his parents took great effort in educating him and helping him build career goals. He showed excellent academic performance in middle school and entered Tianjin University of Science and Technology, Tianjin, China in 2011 and earned his bachelor’s degree in 2015. Then he started to pursue a master’s degree of Bioinformatics at The University of Texas at El Paso (UTEP). In 2017, he got his master degree and joined Ph.D. Program of Computation Science (CPS) at UTEP. During this period, he has been working as a Teaching Assistant in the Department of Mathematical Sciences and Research Associate in the School of Pharmacy. In 2018, he was elected as the president of Chinese Student and Scholar Association. In 2019, he earned his second Masters degree in CPS and was awarded a Big Data Analytics certificate. Then he continued his study in CPS and now is approaching completion of the PhD degree.