Optimizing large scale omics analyses with data management techniques

Candidate Number: 1052483

University of Oxford

A thesis presented for the degree of

Master of Science in Computer Science

Trinity 2021

Acknowledgements

Acknowledgments were omitted for anonymization.

Abstract

The use of machine learning techniques for analyzing biomedical data has seen a big increase in recent years due to promising results and various innovations achieved. However, the nested structure of biomedical data still remains a major issue when it comes to translating data to a format compatible with languages prevalent in machine learning, such as Python. In addition, the large size of data often results in executing the processing of data using distributed techniques. This, however, brings additional challenges for nested data. In this thesis, we extend a nested query compilation framework by incorporating statistical operations in the framework, integrating User-Defined Functions, and exploring different feature selection methods to improve the performance of models on various tasks regarding cancer prediction.

Contents

List of Figures

List of Tables

List of Abbreviations

NRC	Nested Relational Calculus
TraNCE	The extended Nested Relational Calculus Framework
TCGA	The Cancer Genome Atlas
GMB	Gene Mutation Burden
UDF	User-Defined Function
RFE	Recursive Feature Elimination
ANOVA	Analysis Of Variance
GRCh37	Genome Reference Consortium Human Genome build 37

1 Introduction

1.1 Motivation

Biomedical statistical inference can have a huge impact on the future of medicine. While in this thesis we will not focus on hypothesis testing and causal statistics which was the main area of research in the last years [35, 28], we will investigate how statistical operations when incorporated in query language compilation frameworks can help our understanding in cancer prediction tasks. We will discuss challenges surrounding data processing of complicated biomedical data and their analyses and present a novel query compilation framework, TraNCE, developed by Smith J. et al. [38]. We will also discuss the importance of UDFs in query languages and tackle the challenging task of optimizing such functions.

In this work, we look to extend an existing nested compilation framework with User-Defined Functions (UDFs). We use case studies from Machine Learning (ML) based biomedical analyses to explore UDF optimizations specific to feature selection and evaluate the optimizations with respect to both runtime and model accuracy.

1.2 Thesis Objectives

In summary, the project will focus on the following tasks:

- The extension of NRC with UDF support as a means to apply non-trivial statistical operations in NRC
- The extension of the shredding transformation incorporated in TraNCE to support UDFs
- The optimization of UDFs by enabling hints to the users
- The exploration of these extensions with respect to building feature sets and

combining them with multi-omics data analyses approaches and neural networks to perform classification tasks

- The analysis of gene sets from the resulting feature outputs of models from various experiments using gene enrichment analysis tools

1.3 Thesis Outline

In Section 2 we introduce the background material. We first talk about multi-modal biomedical analyses and their importance in cancer-related tasks. We then introduce various data sources we will use in the thesis and discuss data processing platforms and the challenges associated with the parallel execution of tasks. We then introduce TraNCE at a high-level and NRC. Finally, we introduce UDFs; their uses in query compilation frameworks, and the challenges associated with optimizing them. Finally, we discuss feature selection methods and present a detailed discussion for the ones we will be using in the thesis.

In Section 3, we describe how we extended the TraNCE framework by integrating UDFs. We describe how the UDFs were defined, their uses, a hint parameter that helps optimize them and we go into detail about the syntax of the implementation in TraNCE.

In Section 4, we describe how we used hints to enable the pushing of filters within UDFs with the optimizations being specific to feature selection in ML tasks. We also discuss the methods of two filters that were provided and integrated into the TraNCE framework.

In Section 5 we present the experiments we considered for the binary and multi-class classification tasks, the experimental setup, the results, and some discussion on the results. We also talk about gene enrichment analysis and present findings on gene sets extracted from the different models we used. In Section 6, we discuss some of the limitations and findings from our experiments and recommend areas for future work, while in Section 7 we present our final conclusions.

2 Background

The focus of the thesis is the biomedical analyses of cancer-related prediction tasks, using multi-modal analyses. Our work uses a nested query compilation framework and aims to extend it to support UDFs. In this section, we first provide details on the biomedical application domain, paying specific attention to multi-modal analyses in Section 2.1 and introduce the data sources we will use throughout the thesis in Section 2.2. We then discuss distributed processing platforms in Section 2.3 and the challenges that arise when executing complex, biomedical analyses in a distributed setting. Furthermore, we introduce the nested query compilation framework, *TraNCE*, in Section 2.4. We continue with a discussion of UDFs - what the state-of-the-art in UDFs is, the role of UDFs in biomedicine, and in the context of processing nested queries. Finally, in Section 2.6, we discuss feature selection methods.

2.1 Multi-modal Biomedical Analyses

Mutational burden and multi-omics cancer driver genes analysis will be the two main tasks investigated. Mutational burden is a generic characterization of tumorous tissue, that can be informative for cancer therapy [31, 42]. *Tumor Mutational Burden* (TMB) is defined as the total number of somatic mutations present in a tumor sample, which is one of the important measures when it comes to cancer research and treatment. *Gene Mutational Burden* (GMB) is a sub-calculation of TMB that defines the total number of somatic mutations in a gene for every tumor sample. The results of the GMB analyses are used as features vectors for various classification and prediction problems. GMB analyses are beneficial for prediction tasks such as predicting the tumorsite of the origin of cancer. Furthermore, there are also other tasks related to specific types of cancer. For example, Z. Chen et al. [25], present a study on using gene expression (see Section 2.2 for details about gene expression data source) for predicting the severity of prostate cancer. TMB analysis, however, involves one data source type. Genomic sequencing, advancements in ML techniques, and the vast amounts of medical data have resulted in interesting ideas surfacing in the biomedical field such as integrating complex datasets. Multi-omics or multi-modal analytics is a biological analysis approach where the data at hand consists of multiple omics, such as the *genomics*, *proteomics* and *transcriptomics*. Genomics is the interdisciplinary field of biology that focuses on the structure and evolution of genomes, which is an organism's complete set of DNA. Proteomics is the large-scale study of proteins, which are vital parts of living organisms. Transcriptomics is the study of all RNA [1] molecules in a cell. The integration of different omics datasets has proven to be vital in understanding human health and diseases, while multi-omics analyses have revolutionized the fields of biology in recent years [23, 13, 39, 20, 8]. The use of omics data became also famous for pathway analysis tasks. K. Ramanan et al. [14], discuss the increasing use of genome-wide data sets for identifying biological pathways for complex diseases, while I. Menashe et al. [10], talk about specific use of genomes for pathway analysis for breast cancer. The integration of omics datasets helps provide better insights to the analysis as a whole since using more data to tackle such challenging tasks [2] is beneficial. In the thesis, we extend TMB analyses by combining different data sources for cancer prediction tasks. Moreover, we also extend one-class cancer prediction models to multi-class models that given a list of different cancer types, attempt to predict the correct one.

These types of multi-omics analyses have lead to an increase in biomedical data collection. Many publically available data sources are now available for the community to explore, such as The Cancer Genome Atlas (TCGA) [19], the 1000 Genomes Project [21], and the UK Biobank [51]. TCGA is a landmark cancer genomic program contain-

---

[1] Ribonucleic Acid, a molecule similar to DNA that is copied from pieces of DNA and contains information to make proteins and perform other important functions in the cell

[2] Such as cancer prediction

ing different omics data with the aim of improving the ability of people to diagnose and treat cancer. 1000 Genomes is an international project containing the most detailed catalog of human genetic variation. The UK Biobank is a large biobank in the United Kingdom investigating contributions of genetic predisposition and environmental exposure to the development of a disease. These datasets are large-scale, multi-omics, and pan-cancer data sources that come in a variety of domain-specific formats. Pan-cancer data sources refer to data sources containing information for multiple cancer types.

2.2 Data Sources

Nested data are data sources with attributes which are themselves nested collections of data on various levels [3]. Nested data are particularly prevalent in many domains including biomedicine because of the complicated structure of data in the field. Relations existing within relations are a very common phenomenon in biomedicine, especially when dealing with gene and gene expression data sources. Gene expression is the process by which information from a gene is used. *Flat* data are data sources having all attributes of scalar type (integer, double, etc.). Somatic mutations are changes to the DNA sequence of a somatic cell. A format common for somatic mutations is the Mutation Annotation Format (MAF), which is a file containing the aggregated mutation information from patients. MuTect is a method that applies a Bayesian classifier to detect somatic mutations with very low allele fractions [18]. The Variant Effect Predictor (VEP) annotates information from somatic mutations [22], returning a collection of occurrences. MAF files produced from MuTect are then annotated with VEP files to create the `occurrences` data source. Prad is a subset of the TCGA dataset that contains somatic mutations of prostate cancer patients only. In addition to the `occurrences` data source, we present below some other real-world data sources that

---

[3] Levels here refers to the depth of nesting

we will be using throughout the thesis, that are a subset of the multi-modal biomedical resources used in integrated analyses.

2.2.1 `occurrences`

Somatic mutation occurrences, `occurrences`, data source is from the TCGA dataset. We present the type of `occurrences` below:

$$\{\ \langle\ \texttt{donorId} : \mathit{string}, \texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{int}, \texttt{reference} : \mathit{string},$$
$$\texttt{alternate} : \mathit{string}, \texttt{mutationId} : \mathit{string}, \texttt{candidates} :\ \{\ \langle \texttt{gid} : \mathit{string},$$
$$\texttt{impact} : \mathit{string}, \texttt{sift} : \mathit{real}, \texttt{poly} : \mathit{real}, \texttt{consequences} :\ \{\ \langle \texttt{conseq} : \mathit{string}\ \rangle\ \}\ \rangle\ \}\ \rangle\ \}$$

2.2.2 `samples`

`samples` comes from the TCGA dataset. It is a data source that returns cancer samples. For simplicity, we assume that it only returns the sample identifier, `sid` and an attribute that acts as an identifier for a single biological sample taken from a patient, `aliquot`:

$$\{\ \langle\ \texttt{sid} : \mathit{string}, \texttt{aliquot} : \mathit{string}\ \rangle\ \}$$

2.2.3 `genemap`

`genemap` comes from the Genome Reference Consortium Human genome build 37, (GRCh37). Reference Genomes are sequences of nucleotides that describe the set of genes from an organism of a species. `genemap` is a data source that provides information about the gene. We provide the type below:

$$\{\ \langle\ \texttt{contig} : \mathit{string}, \texttt{start} : \mathit{int}, \texttt{end} : \mathit{string}, \texttt{name} : \mathit{string}, \texttt{gid} : \mathit{string}\ \rangle\ \}$$

2.2.4 `clinical`

`clinical` data source comes from the TCGA dataset. It provides information about the clinical records of a patient. We present the data type below:

$$\{\ \langle\ \mathtt{sid} : \mathit{string}, \mathtt{gender} : \mathit{string}, \mathtt{race} : \mathit{string}, \mathtt{ethnicity} : \mathit{string}, \\ \mathtt{histtype} : \mathit{string}, \mathtt{tumorsite} : \mathit{string}\ \rangle\ \}$$

2.2.5 `gene_expression`

`gene_expression` data source comes from the TCGA dataset. Gene expression is the process by which the information encoded in a gene is used to assemble a protein molecule. For example `gene_expression` provides information on sequencing RNA. We present the type below:

$\{\ \langle\ \mathtt{aliquot} : \mathit{string}, \mathtt{gid} : \mathit{string}, \mathtt{fpkm} : \mathit{real}\ \rangle\ \}$

To give an example of nested data and the difference between flat and nested data we consider the two data sources `occurrences` and `samples`. `samples` has two attributes, and both of them return a scalar type value (string). `occurrences` though has an attribute `candidates` which itself is a nested collection as it returns five more attributes. The attribute `consequences`, has itself another attribute, called `conseq`. We can see then the depth of nesting is two for `occurrences`.

2.3 Distributed Processing Platforms

Apache Spark [12], Apache Flink [9], and Apache Hadoop [7], are some of the distributed data processing platforms when it comes to processing large-scale data. These platforms provide collection Application Programming Interfaces (APIs) which enable

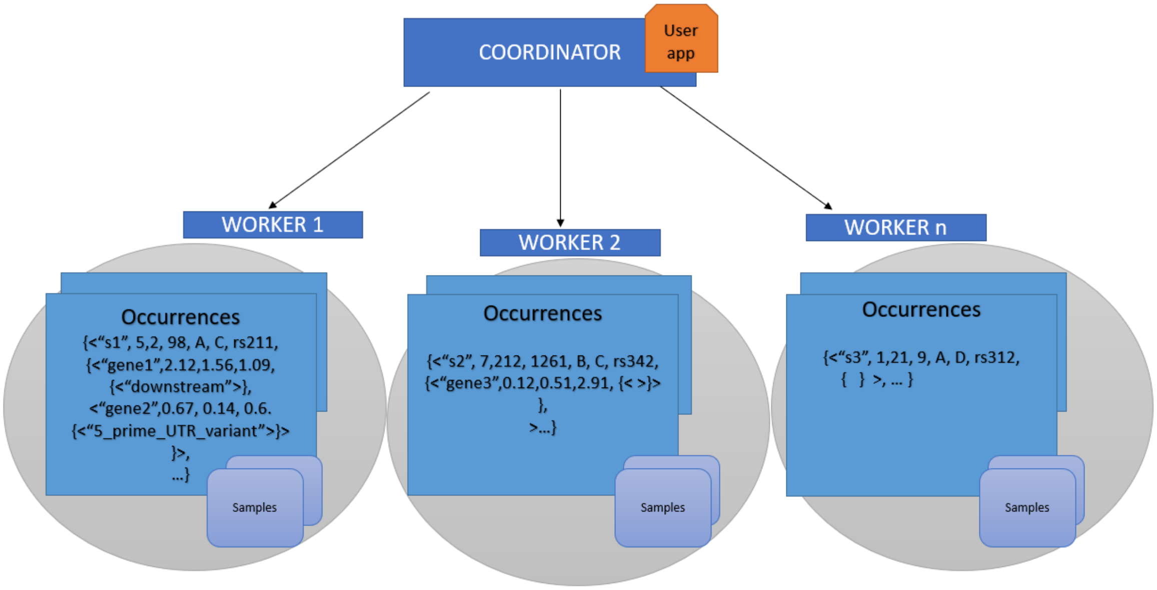

COORDINATOR
User app
WORKER 1
WORKER 2
WORKER n
Occurrences
{<"s1", 5,2, 98, A, C, rs211,
{<"gene1",2.12,1.56,1.09,
{<"downstream">},
<"gene2",0.67, 0.14, 0.6.
{<"5_prime_UTR_variant">}>
}>,
...}
Samples
Occurrences
{<"s2", 7,212, 1261, B, C, rs342,
{<"gene3",0.12,0.51,2.91, {< >}>
},
>...}
Samples
Occurrences
{<"s3", 1,21, 9, A, D, rs312,
{ } >, ... }
Samples

Figure 2.1: Example of a distributed representation of a user-defined application. `occurrences` are cached in memory across the n worker nodes.

programmers to define complex analytical tasks involving distributing resources and data parallelism in an abstract way. Since these platforms use collection APIs they also support nested data, however, the distribution strategies often fail to process nested collections due to top-level distribution strategies. These distribution strategies ensure that all nested collections are on the same machine as their parent, which can lead to distribution issues especially with large inner collections or few top-level tuples. Moreover, data scientists are often faced with difficulties when translating analyses into distributed settings. Below follows an explanation of such distributed data processing platforms using Spark as a running example.

There is one central *coordinator node* while the rest of the nodes are *worker* nodes. Figure 2.1 shows a simple application being executed by the Spark cluster: First, the coordinator node receives the task and then delegates the tasks to the worker nodes in a distributed way.

Spark represents distributed data using Resilient Distributed Datasets (RDDs). RDD is an immutable distributed collection of elements of the data partition across the available nodes in the cluster, operating in parallel [15]. When flat data sources (such as `samples`) are imported into the Spark cluster, they are allocated in a round-robin fashion across all available partitions. If a data source is nested (such as `occurrences`), the difference is that the nested attributes remain in the same partition as their parent This is the top-level distribution strategy. Figure 2.1 displays exactly this: `samples`, are stored in memory across the worker nodes distributing the top-level attributes while keeping the nested attributes in the same location.

Many challenges surface for programmers when writing programs over distributed, nested collections, with the most important being the bottleneck created by the (often) few top-level attributes. For example, grouping impact scores (from the `clinical` example from before) represented by a small number of tumorsite tissue, results in a distribution not exceeding the number of tumorsite tissues. This is problematic since the full potential of the cluster is not used. Moreover, when inner collections are large, the process of caching in and out of memory can strain the physical storage thus resulting in skew-related bottlenecks. Finally, when joining on nested attributes, the nested attributes are nested within each partition that their parent attribute is allocated to and hence cannot be accessed without iterating in the nested collections. Since the partitions are not aware of the contents of each other, the keys cannot be referenced outside the partitions. A naïve solution would be to copy the parent data source to each worker node, however that is usually very computationally expensive.

One of the aims of the project is to integrate multiple complex (nested) datasets (which are large) because of the benefits they proved to bring when approaching cancer-specific biomedical tasks. Integrating such datasets and executing data handling procedures in a distributed fashion is a non-trivial task. We now present a query compilation framework, TraNCE that provides solutions for the above challenges.

2.4 TraNCE

In this section, we describe a nested query compilation platform, TraNCE, developed by Smith J. et al. [38], that overcomes the challenges of processing nested data on distributed platforms. The framework performs query compilation using a variant of Nested Relational Calculus (NRC). This overcomes the difficulties of programming with a distributed collection API on nested datasets. TraNCE also uses a specialized data representation to address distribution challenges associated with nested datasets. This section will begin with an overview of TraNCE at a high-level, introducing the architecture and the components available for handling the complexities of distributed, nested data transformations. In Section 2.4.4 we present an example of a program, called GMB, and the high-level language associated with it. We then describe how NRC is translated into a query plan and compiled into executable Spark code, known as standard compilation. We then describe the shredded compilation route that is optimized for handling nested data in a distributed setting in Section 2.4.5 and give a detailed breakdown of the shredded transformation of the GMB program. In Section 2.4.6 we present a high-level overview of the pipeline of the mutational burden analyses and how some challenges are tackled by the TraNCE framework.

2.4.1 TraNCE overview

Modern biomedical analyses are essentially pipelines of different data access mechanisms that work on and produce datasets of varying complexity [40, 11]. Integrating such complex datasets remains a challenging task for programmers, especially for processing large-scale data. Advances in genomic sequencing and medical data management have produced large-scale datasets whose integration creates many problems for existing data processing platforms, such as Apache Hadoop, Apache Flink, and Apache Spark.

TraNCE is a compilation framework that transforms declarative programs (a method of building the structure of programs that expresses the logic of a computation without describing its control flow) over nested collections into distributed execution plans. The framework can be broken down into three key aspects:

- Program compilation
- Program and data shredding
- Skew-resilience

Program compilation makes use of a high-level, declarative language allowing users to describe programs over nested collections without burdening them with the problem of handling nested collections in distributed settings.

The TraNCE framework provides two compilation routes: *standard* and *shredded*. The standard compilation makes use of unnesting [4], techniques in order to apply flattening methods to handle nested values while the shredded compilation optimizes the standard compilation by using *shredding* techniques. The shredding techniques transform a program that operates on nested collections into multiple programs that operate on flat collections [3], with the advantage being that parallelism is allowed beyond top-level records. The output of both compilation routes is an Apache Spark program, defined using the Scala API, that can be used in a distributed setting.

Even when dealing with flat data only, data skew can highly affect the performance of large-scale processing of data. Data skew sometimes leads to some worker nodes having a bigger workload than other workers leading to longer run times of programs. Nested data aggravates this problem since inner collections might have skewed cardinalities (an example is a join on a small number of patients having a big number of medical records). *Skew-resilience* prevents such overloading of partitions thus maintaining a better distribution of data across the worker nodes.

2.4.2 Architecture

Here we overview the TraNCE architecture and outline the steps in how TraNCE compiles the source language (NRC) into generated (Spark) code through a series of transformation steps. The transformation steps are summarized in the flow diagram below in Figure 2.2. The high-level language is described in more detail in Section 2.4.4. The full syntax of the TraNCE language is provided in [37].

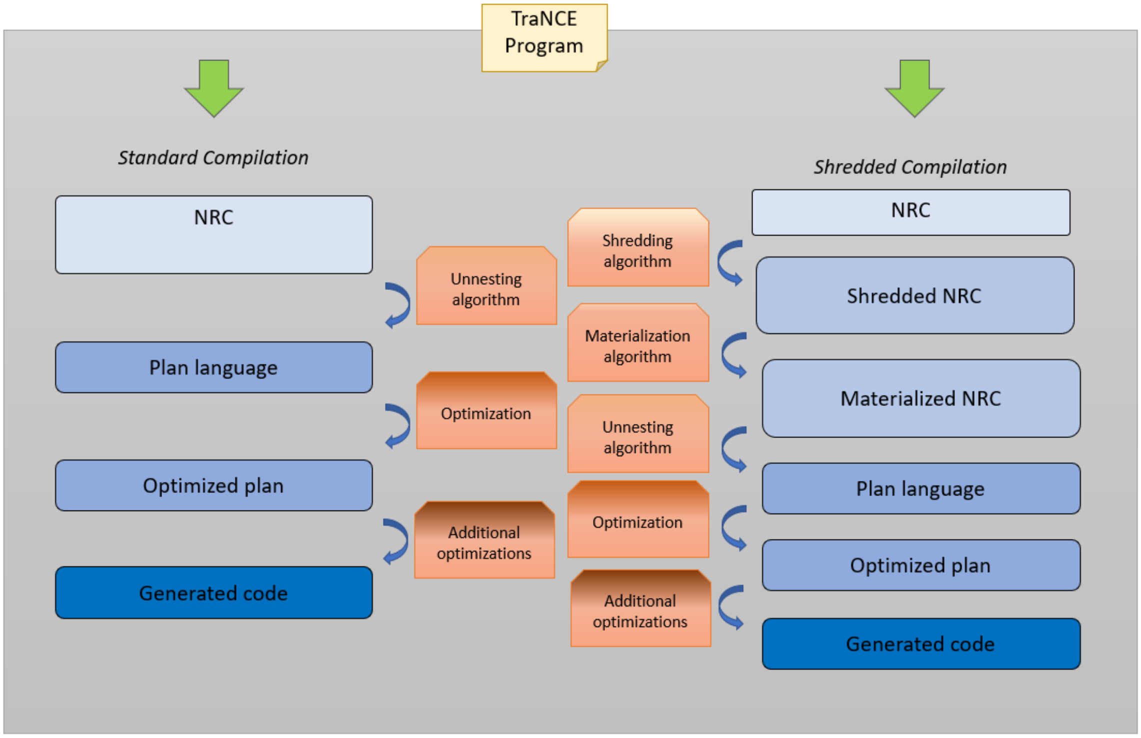


Figure 2.2: Transformation steps from raw input query string to generated code. Each language is a set of Scala case classes and each transformation step is an algorithm consisting of pattern matching.

When a source query is submitted to the program it can be compiled through a standard or shredded compilation route. The standard compilation route translates a source NRC query into a query plan. The query plan is then optimized and passed through the code generator to produce an executable Spark application. TraNCE provides support to compile out to Apache Zeppelin [24] notebooks. One of the reasons why we use Zeppelin notebooks is because they support various code interpreters. The shredded compilation route extends the standard compilation route by providing an alternative, more succinct representation to nested data. The output of the shredding steps (query shredding and materialization shown in Figure 2.2 above) will then proceed with the rest of the compilation route, translating shredded NRC into executable Spark code that operates on this succinct representation. Section 2.4.5 describes the shredded compilation route by example. The code can be found here [49].

2.4.3 NRC

NRC is a declarative database query language that is an extension to relational calculus and is more suitable for relational models dealing with nested-structured data and thereby nested queries [12, 2]. The syntax of NRC that is used in the TraNCE framework is given below in Figure 2.3.

$$
\begin{aligned}
P &::= (var \Leftarrow e)^* \\
e &::= \emptyset_{Bag(F)} \mid \{e\} \mid \texttt{get}(e) \\
&\quad \mid c \mid var \mid e.a \mid \langle a_1 := e, \ldots, a_n := e \rangle \\
&\quad \mid \texttt{for}\ var\ \texttt{in}\ e\ \texttt{union}\ e \mid e \uplus e \\
&\quad \mid \texttt{let}\ var := e\ \texttt{in}\ e \mid e\ PrimOp\ e \\
&\quad \mid \texttt{if}\ cond\ \texttt{then}\ e \mid \texttt{dedup}(e) \\
&\quad \mid \texttt{groupBy}_{key}(e) \mid \texttt{sumBy}^{value}_{key}(e) \\
cond &::= e\ RelOp\ e \mid \neg cond \mid cond\ BoolOp\ cond \\
T &::= S \mid C \\
C &::= Bag(F) && \textit{– Collection Type} \\
F &::= \langle a_1 : T, \ldots, a_n : T \rangle \mid S && \textit{– Flat Type} \\
S &::= int \mid real \mid string \mid bool \mid date && \textit{– Scalar Type}
\end{aligned}
$$

Figure 2.3: Syntax of `NRC`.

We now explain the grammar of the NRC syntax from Figure 2.3 in some detail.

P denotes a program, which is a sequence of assignments of *variables*, *var*, where each variable is an *expression*, e. Aggregation and deduplication are enabled through the expression language, thus extending the standard NRC. The standard NRC types constitute of the basic scalar types (e.g. integer and string types), tuple types, $<a_1{:}T_1 \ldots a_n{:}T_n>$ and bag type, *Bag*, T. In the version of NRC used by TraNCE, the contents of the bag type are restricted to be a tuple or scalar type.

RelOp denotes the comparison operator on scalar type variables (e.g. $<$,$>$), *PrimOp* denotes a primitive function on scalars (e.g. $-$, $*$) and *BoolOp* denotes a boolean operator (e.g. $||$, &&). Variables can be free or part of a `for` or `let` constructs. e returns a singleton bag from an expression, and `get`(e) takes as input a singleton bag and returns its only element; in the case where e is empty (or has more than one element) a default value is returned. $\varnothing_{Bag(F)}$ simply returns the empty bag. `dedup`(e) takes as input a bag e and returns a bag with the same elements, but with all multiplicities (number of occurrences of each elements) being one. $\texttt{groupBy}_{key}$(e) groups the tuples of bag e using a collection of attributes *key*, and for each distinct *key* it produces a bag named GROUP that contains the projected tuples from e with the *key* value. $\texttt{sumBy}^{value}_{key}(e)$ first groups the tuples of bag e by the values of their *key* attributes whereas for each distinct value of *key* it sums up the attributes *value* of the tuples with the *key* value.

2.4.4 Example TraNCE Program

We provide a walk-through of the high-level language using an example program that we call `GMB`:

```
GMB ⇐ for g in genemap union
    {⟨ gene := g.name, burdens := sumBy^burden_sid(
      for o in occurrences union
        for t in o.consequences union
            if g.gid == t.gid then
                {⟨ sid := o.sid, burden :=
                (if (t.impact = "HIGH") then 0.80
                else if (t.impact = "MODERATE") then 0.50
                else if (t.impact = "LOW") then 0.30
                else 0.01)⟩})⟩}
```

TraNCE has the advantage of returning results with nested output type while abstracting the complexities of nested distribution of data from the user. Considering the `GMB` program, the $\Leftarrow$ operator requests specific attributes from the `genemap` data source. The program iterates then over the top-level of `genemap` keeping the `gene` attribute. It then creates a nested `burden` collection using the `sumBy` $^{burden}_{sid}(e)$ function. `sumBy` can be applied at a specified level given that the input e is not a nested collection itself. The `sumBy` function is applied on the top-level of `occurrences` with *sid* as the key and *burden* as the value and since all attributes are of scalar type the input (equivalent to e) has a flat type:

$$\{ \langle \texttt{sid} : \mathit{string}, \texttt{burden} : \mathit{real} \rangle \}$$

Inside `sumBy`, first the `occurrences` data source is introduced, then the program iterates over the `consequences` attribute of `occurrences` and the nested *burden* collection is created only on the common `id` attributes of the two datasources. Finally, the iteration preserves only the `sid` and `impact` attributes. The final output of the program is then:

$$\{ \langle \texttt{name} : \mathit{string}, \texttt{burdens} : \{ \langle \texttt{sid} : \mathit{string}, \texttt{burden} : \mathit{real} \rangle \} \rangle \}$$

2.4.5 Shredded Compilation

As mentioned earlier, we will focus on the shredded compilation. Details about the standard compilation can be found here [37]. The advantage of using the shredded pipeline is that scalability of processing data is ensured. To see this, we outline the breakdown of the shredded compilation.

The first phase of the shredded compilation route is program shredding. First, programs are transformed using the *shredded transformation* which transforms programs working on nested data into a set of programs operating on flat data, called *shredded programs*. The inputs are hence expected to be flat relations, called *shredded inputs*. Attributes of both shredded inputs and programs that correspond to nested collections are referenced using *labels*. Labels reassociate the levels of the shredded inputs. The program shredded representation consists of data sources of type `Top`, for the top-level source, type `Dict_attribute_1`, for the first-level source, `Dict_attribute_1_attribute_2` for the second-level source and so on.

The second phase of the shredding algorithm is materialization. Materialization translates the *symbolic representation* to the shredded program. The symbolic representation is defined on output dictionaries and is a partial function translating labels to bags. The framework uses an intermediate query language, $NRC^{Lbl+\lambda}$ that extends NRC with a new atomic type for labels and a function type for dictionaries. We present below in Figure 2.4 an overview of the shredded compilation, while a high-level overview of the materialization algorithm can be found in the original paper [38].

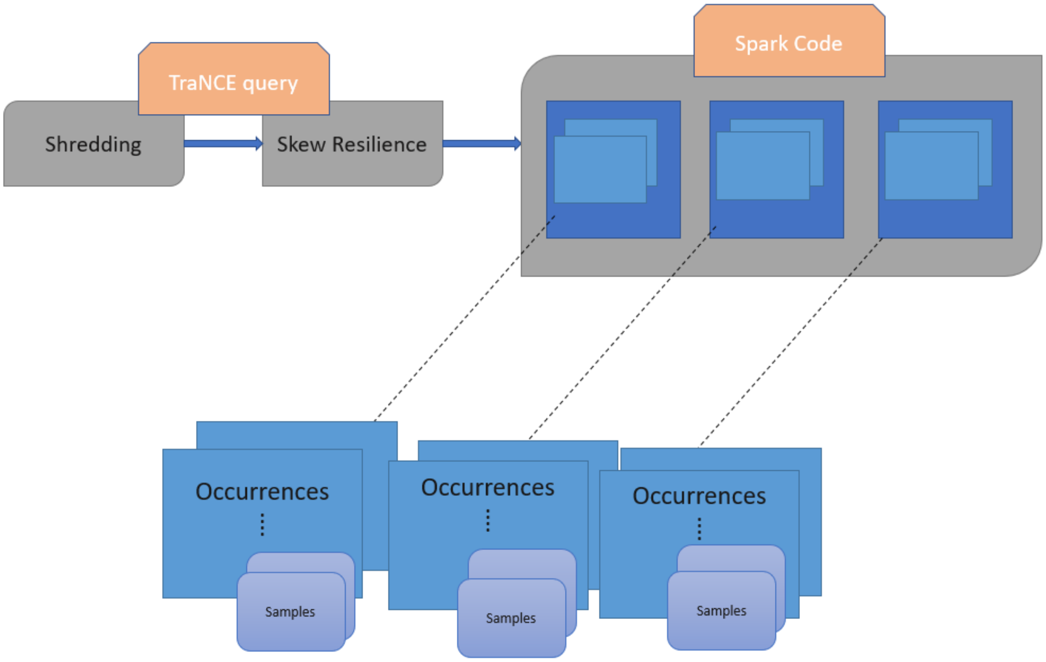


Figure 2.4: Architecture of TraNCE showing the shredded compilation route. Spark provides a schematic representation of the route, while the shredded inputs of `occurrences` are cached in memory across worker nodes.

The shredded compilation provides a succinct representation of data - this representation replaces lower and upper-level attributes with labels. Shuffling is a process of redistributing data across partitions. Data shuffling in combination with the succinct representation results in a reduced data transfer thus providing support for local operations (operations directly applied in the program). We provide an example of the shredding procedure below.

Consider the data source `occurrences` from Section 2.2. The shredded representation of `occurrences` consists of three data sources:

- A top-level source, `occurrences_top` which returns data with a flat type:

$$\{\ \langle\ \texttt{sid} : \textit{string}, \texttt{contig} : \textit{string}, \texttt{start} : \textit{int}, \texttt{end} : \textit{int},$$
$$\texttt{reference} : \textit{string}, \texttt{alternate} : \textit{string},$$
$$\texttt{mutationId} : \textit{string}, \texttt{candidates} : \textit{Label}\ \rangle\ \},$$

- The first-level data source, denoted by `occurrences_Dict_candidates` that is of flat datatype and extends the type `candidates` with a label attribute of type *Label*:

$$\{\ \langle\ \texttt{label} : \textit{Label}, \texttt{gene} : \textit{string}, \texttt{impact} : \textit{real},$$
$$\texttt{sift} : \textit{real}, \texttt{poly} : \textit{real}, \texttt{consequences} : \textit{Label}\ \rangle\ \},$$

- The second-level data source that extends the type of `consequences` with a label attribute of type *Label*, denoted by `occurrences_Dict_candidates_consequences`:

$$\{\ \langle\ \texttt{label} : \textit{Label}, \texttt{conseq} : \textit{string}\ \rangle\ \}.$$

The labels could also be conceptualised as foreign-key dependencies. The `consequences` attribute in `occurrences_top` is a foreign key that references the primary key of `occurrences_candidates`, `label`.

We now provide an example of the shredded compilation route using the program defined in Section 2.4.4, `GMB`. The shredding transformation returns two programs, where together they represent the shredded `GMB` program. The first program represents the top-level collection, namely **`GMB_top`** depicted below:

```
GMB_top ⇐ for g in genemap_top union
    {⟨ gene := g.name, burdens := NewLabel(⟨ gene := g.name⟩ ) ⟩}
```

The type of **`GMB_top`** is:

$$\{ \, \langle \, \texttt{gene} : \mathit{string}, \texttt{burdens} : \mathit{Label} \, \rangle \, \}.$$

We can see that there are no nested collection attributes, and that this is a flat collection. Also, the label attribute of burdens holds only the necessary information to reconstruct the nested output. The program **`GMB_burdens_Dict`** represents the succinct representation of the first-level expression, shown below:

```
GMB_burdens_Dict ⇐ $\texttt{sumBy}_{sid}^{burden}$(
      for o in occurrences union
        for t in o.consequences union
            if g.gid == t.gid then
                {⟨label := NewLabel({⟨ gid := t.gid}⟩ , sample := o.sid, burden :=
                (if (t.impact = "HIGH") then 0.80
                else if (t.impact = "MODERATE") then 0.50
                else if (t.impact = "LOW") then 0.30
                else 0.01) ⟩} ⟩})
```

2.4.6 Pipeline Overview

Returning back to the mutational burden calculation in the TraNCE framework prior to our work, we present in Figure 2.5 below the workflow of the burden-based analysis.

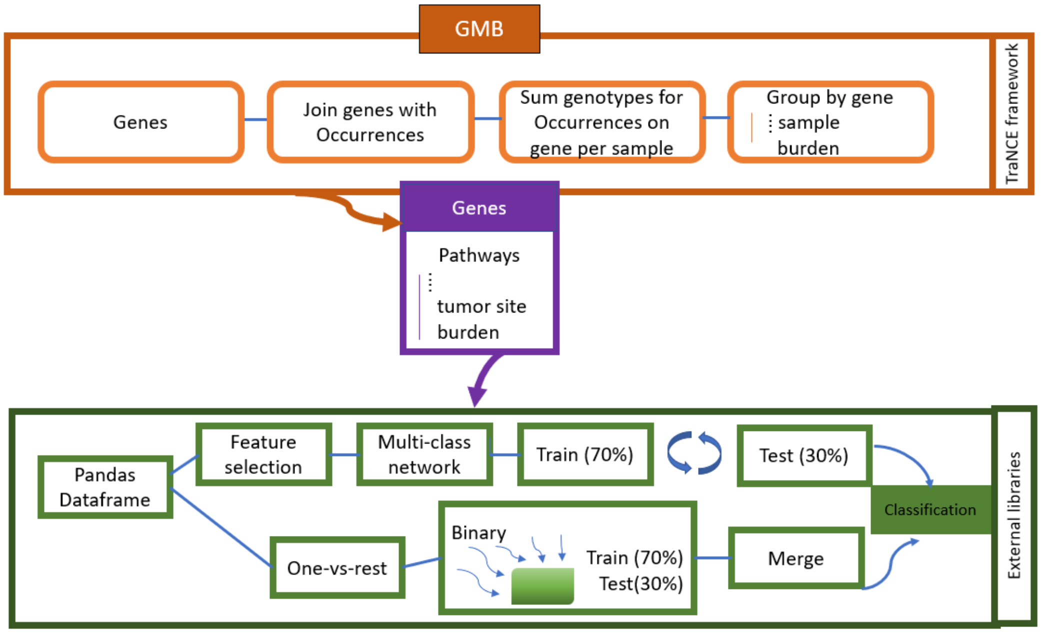


Figure 2.5: Workflow diagram representing the burden-based analysis for gene pathway and the downstream classification problem. The results of the burden analysis are fed into a model of the multi-class or one-vs-rest classification method to predict tumor of origin.

Initially, an NRC program (for this example, the GMB program defined in Section 2.4.4) is defined. The fact that the initial program defined in NRC is extended through the TraNCE framework (since TraNCE supports the transformation of declarative programs) to nested collections and then into plans for execution in a distributed setting, helps tackle the challenges associated with distributed computing defined in Section 2.3. In particular, it eases the burden for programmers to optimize aggregations (such as pushing joins to avoid the bottleneck created by instances of queries with few top-level attributes) as this process is automated via the TraNCE framework. The shredded transformation in distributed settings is also very important in overcoming the challenges discussed earlier. First, the shredded representation of program inputs enables full parallel processing of large nested collections of data and avoids data skew by evenly distributing data across worker nodes. Second, shredding saves computational cost as it is data-efficient - it eliminates the need of reconstructing intermediate nested results. While TraNCE enables distributed computing and tackles data skew problems, there are still additional aspects of this pipeline that require the use of external statistical libraries as we can see in Figure 2.5 above. The usual workflow of relational databases involves queries/programs defined in the query language and then external functions, called UDFs, that handle downstream training - we introduce UDFs below.

2.5 UDFs

In this section, we will discuss the concept of User-Defined Functions (UDFs). The second part of the pipeline in the previous section, Figure 2.5 focuses on performing classification analysis on the feature vectors constructed from an upstream TraNCE program. During this stage, the output of the query is transformed into a Pandas Dataframe, with the transformation being done in external libraries. We denote this as a UDF and use this as a motivating example throughout the rest of this thesis. We now provide an overview of what UDFs are, including the complexities they introduce in the optimization of declarative queries, such as those defined in NRC.

UDFs are functions that define data transformations that can be used in data analysis but cannot be expressed in the query language. With the size and complexity of data rising in recent years, understanding such functions becomes an ever more challenging task. Analyses of complex data are usually described as dataflows in declarative dataflow languages [26]. To achieve scalability for such analyses, the most practical way is to use UDFs to perform various tasks, such as classification or clustering, instead of optimizing the declarative language programs directly. This, however, has as a downside that UDFs end up being “black-boxes” with the user being unable to either optimize or analyze the UDFs. In the context of dataflows, some researchers have presented a way of turning UDFs into “grey-boxes” by exploiting their semantic properties with the help of user annotations [16]. UDFs in the context of biomedicine might range from programs for non-parametric fitting of such functions to plot results from experimental data, to online tools to be used in a transparent way by the user as part of a database query [1, 5]. In the thesis, since TraNCE does not currently support UDFs, we will first define UDFs for particular tasks and implement them in the framework for a more efficient and streamlined end-to-end analysis. Moreover, we will provide user hints to explore how we can apply optimizations. UDFs being “black-boxes” in the TraNCE framework means that the framework has no control over them and that they will be defined externally (outside the framework). In our work, even though UDFs will still remain “black boxes” outside the query language, we explore how we can apply optimizations.

As an example, consider an NRC program (such as the one in Section 2.4.4). The output of the program will be used to build the feature vector and this happens within TraNCE. However, consider a simple UDF defined to train a model (a neural network for example) - `myudf`(*dataset*, $features$, *cancertype*). The user can input which dataset to use, which features to input to the model, and which cancer type to predict. Even if `myudf` is implemented in TraNCE, it still remains an external function that cannot be controlled by the framework. What we will investigate is how we can push optimizations in the framework by defining hints for the user and therefore how to use the UDFs internally within the framework.

We now introduce and describe a common UDF in biomedical classification problems, the concept of *feature selection*.

2.6 Feature Selection

Feature selection is a very important concept related to biomedical data mining, and in particular when dealing with omics data. Due to the complex structure of omics data, it is common for datasets to lie in very high dimensional spaces. This means that the dataset contains a large number of attributes, and this creates problems of overfitting when those attributes are used as predictors in classification tasks. *Overfitting* occurs when a model learns noise details in the training data hindering the ability of the model to perform well when seeing new data (the test data), especially in cases where the amount of data is not sufficient (while there is not an accepted threshold on what “not sufficient” means, the general consensus across papers is that the number of data points should roughly be 10 times more than the number of features [17, 6]). Hence, the need of reducing the input features when performing classification/supervised learning tasks becomes very significant and necessary. As we will see in the Experiments Section, 5.3.1, where the output number of features was around 55 000, such tasks require a huge amount of data available if we do not perform feature selection, deeming feature selection necessary.

In particular, there are filter, wrapper, and embedded methods. *Filter* methods use a specific metric to identify irrelevant attributes. This metric could be a correlation metric (such as Pearson's correlation), chi-square values, and so on. After identifying irrelevant attributes, redundant columns (i.e. features) are filtered out. Scores for each feature are calculated using the determined metric. Then, the k [4] selected features with the highest scores are chosen (where k is determined by the user) or all the features whose score exceed a user-determined threshold, τ [5]. *Wrapper* methods consider subsets of the set of all the features and for each subset, they fit a supervised model, such as a random forest or a decision tree. The subsets are then evaluated by a performance metric (usually classification accuracy but is dependent on the task) calculated from the resulting model. *Embedded* methods try to combine the qualities of both previous methods by using an implementation of algorithms that have their own built-in feature selection methods (some examples include LASSO and RIDGE regression). In the thesis, we will focus on filter and wrapper methods. Due to the fact that wrapper methods use a predictive model to score a feature subset and each subset is used to train a model, it is expected that they will provide the best performing feature set compared to filter methods. However, this comes with a very heavy computational cost. Filter methods are usually less computationally expensive than wrapper methods, however, they often end up with sub-optimal feature sets. This is because the feature set is not tuned to a specific type of predictive model, and is hence more general than the resulting set of wrapper methods. It is therefore of interest to determine the size of the feature set at which the computational cost of wrapper methods becomes too large for them to be used within an acceptable time frame (although there is no precise definition of what "acceptable" is, we will discuss in next sections

---

[4] $k \in \mathbb{N}$

[5] $\tau \in \mathbb{R}$

how we defined it). Below, we explain in detail how the feature methods we considered work.

FILTER METHODS:

ANOVA

Analysis Of Variance (ANOVA) is a filter feature selection method that uses the concept of variance. The intuition behind ANOVA can be explained using a simple example. Suppose that we have two classes (the labels, or what we want to predict) and two features, x and y, and we want to end up with a score that represents "how well does this feature discriminate between the two classes". If a feature is to be a better separator of the classes, a good metric to use is whether the distance between the mean of class distribution of x is more than y. This procedure can be formalized in the summary below:

- Find the mean from all observations, $\bar{x}$
- Find the mean of all features individually
- Calculate the numerator of the statistic, F:
  $numerator = n_{feature1}(\bar{x}_{feature1} - \bar{x})^2 + n_{feature2}(\bar{x}_{feature2} - \bar{x})^2 + ...$
  $... + n_{featurek}(\bar{x}_{featurek} - \bar{x})^2$
- Calculate the denominator of F, concept similar to the sample variance:
  $denominator =$

  $$\frac{1}{(n_{feature1}-1)+...+(n_{featurek}-1)}\sum_{n=1}^{k}(x_i-\bar{x}_{feature1})^2+...+(x_k-\bar{x}_{featurek})^2$$

- We then calculate the F, $F = \frac{numerator}{denominator}$. F here represents the F-statistic, which calculates the ratio between explainable and unexplainable variance. The higher the score the better discrimination between the classes we want to predict and hence when performing ANOVA we can keep the features with the highest scores. Thus, if we want to select between x or y, we choose the one with the bigger F score.

Chi-square:

A chi-square test is used to test the independence of two events. In prediction tasks, where we want to determine the relationship between the independent feature category (the predictor) and the dependent feature category (label, what we want to predict), ideally, we want to have features that are highly dependent on the response. In summary, the calculation of the χ^2 statistic is as follows:

- Perform a dot product between feature and the response (response should be binarised, i.e. be either 0 or 1)
- Sum over the feature values and calculates class frequency
- Calculate the dot product and gets the expected and observed matrices, (O_k and E_k respectively)
- Calculate the χ^2 value based on the equation:

$$\chi^2 = \frac{1}{d} \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

- Define the hypothesis of the test. The null [6] hypothesis is usually that the two variables, (feature and response) are independent and the alternate [7] hypothesis is that they are not independent.
- Compare the χ^2 obtained from the feature using the χ^2 distribution test (with degrees of freedom being equal to the number of class - 1) to find the p-value

Large χ^2 values (and hence small p-values) indicate that the hypothesis that the two categories (feature and response) are independent is incorrect. In other words, the higher the χ^2 value the more dependent the feature is on the response and so can be selected as a feature

---

[6] The hypothesis that there is no significant difference between the two specified variables, and that any difference we observe is due to sampling or experimental error

[7] The hypothesis stating that something else is happening, usually the negation of the null hypothesis.

to train the model.

Mutual Information:

Mutual information (MI) between two random variables is a non-negative value, which measures the dependency between the variables [27]. It is equal to zero if and only if the two random variables are independent, and higher values imply higher dependency. The function relies on nonparametric methods based on entropy estimation from k-nearest neighbors distances. The MI of variables X and Y for example, I(X,Y) is given by:

I(X;Y) =

$$\int_X \int_Y p(x,y) log \frac{p(x,y)}{p(x)p(y)}\, dx\, dy$$

p(x,y) represents the joint probability density function of X and Y, where p(x) and p(y) represent the marginal density functions of X and Y respectively. MI determines how similar p(x,y) is with the product of the logarithm of the marginal distributions. If X and Y are independent, hence unrelated, then p(x,y) is equal to p(x)p(y) and so the logarithm of $\frac{p(x,y)}{p(x)p(y)}$ would be equal to 0. We therefore want to keep features to be passed for training that maximize the MI.

MultiSURF:

The algorithm MultiSURF is based on the paper by R. Urbanowicz et al. [29]. The algorithm is based on *Relief-based* feature selection methods, referred to as Relief-Based Algorithms (RBAs). RBAs are some of the few filter methods that can capture feature interactions, and specifically gene-gene interactions owing to the use of the "nearest neighbor instances" approach. A difference between RBAs and the other feature selection methods considered in the thesis is that they do not eliminate feature redundancies, i.e. uncorrelated features. Algorithm 1 below shows how MultiSURF operates.

In Algorithm 1, *diff* returns 0 if the value of of the instances is the same, and 1 otherwise,

Algorithm 1: Pseudo-code for the MultiSURF algorithm

```
$n \leftarrow$ number of training instances;
$a \leftarrow$ number of attributes (features) $k \leftarrow$ number of nearest hits, 'H' and
 misses 'M'
STAGE 1
pre-process dataset
STAGE 2
pre-compute distance between all pairs
for i:=1 to $n$ do
    set $T_i$ to mean distances between instance $i$ and all others
    set $\sigma_i$ to standard deviation of those distances
end
STAGE 3
initialize all feature weights, W[A] := 0.0
for i:= 1 to $n$ do
    IDENTIFY NEIGHBORS
    initialise hit and miss counters $h$:= 0.0 and $m$:= 0.0
    for j:=1 to $n$ do
        if distance between i and j is $< T_i$ - $\sigma/2$ (using distance array) then
            if j is a hit then
                h += 1
            else
                m+=1
            end
        end
    end
    FEATURE WEIGHT UPDATE
    for all hits and misses do
        for A:= to $a$ do
            W[A]:= W[A] - $diff$(A,$R_i$,H)/($n$.k) + $diff$(A,$R_i$,M)/($n$.k)
        end
    end
end
return vector W of feature scores that estimate the quality of features
```

while for triples it returns:

$diff(\mathrm{A},I_1,I_2) = \frac{value(A,I_1)-value(A,I_2)}{max(A)-min(A)}$

Hits refer to neighbors having the correct label (that we want to predict), and misses to neighbors having the wrong labels. For more details on the MultiSURF algorithm, we refer the reader to the original paper [29].

WRAPPER METHODS

Recursive Feature Elimination:

Recursive Feature Elimination (RFE) is a wrapper-style feature selection method that also uses filter-based feature selection techniques internally. Given an external estimator that assigns weights to features (e.g. random forest) the goal of RFE is to select features by recursively considering smaller and smaller subsets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained. The importance of each feature is closely related to concepts of collinearity and dependencies that might exist in the model. The "least important" features are then removed from the current set of features. This procedure is then recursively repeated on the smaller set (with the unimportant features removed) until the desired number of features to select (from the input of the user) is eventually reached.

The works of Bommert A. et al. provide useful insights into feature selection methods [36]. While their work focuses on benchmarking different filter methods both in terms of model performance [8] and runtime of filtering, they provide useful insights on the correlation between the different feature selection methods. In particular, they assess the similarity of the filter methods by comparing the order in which they select features for various datasets. They then compute the rank correlation between the orders of all pairs of the filter methods they consider and each dataset. The results in Figure 2.6 below denote the average correlation from all datasets based on the arithmetic mean.

---

[8] Classification accuracy

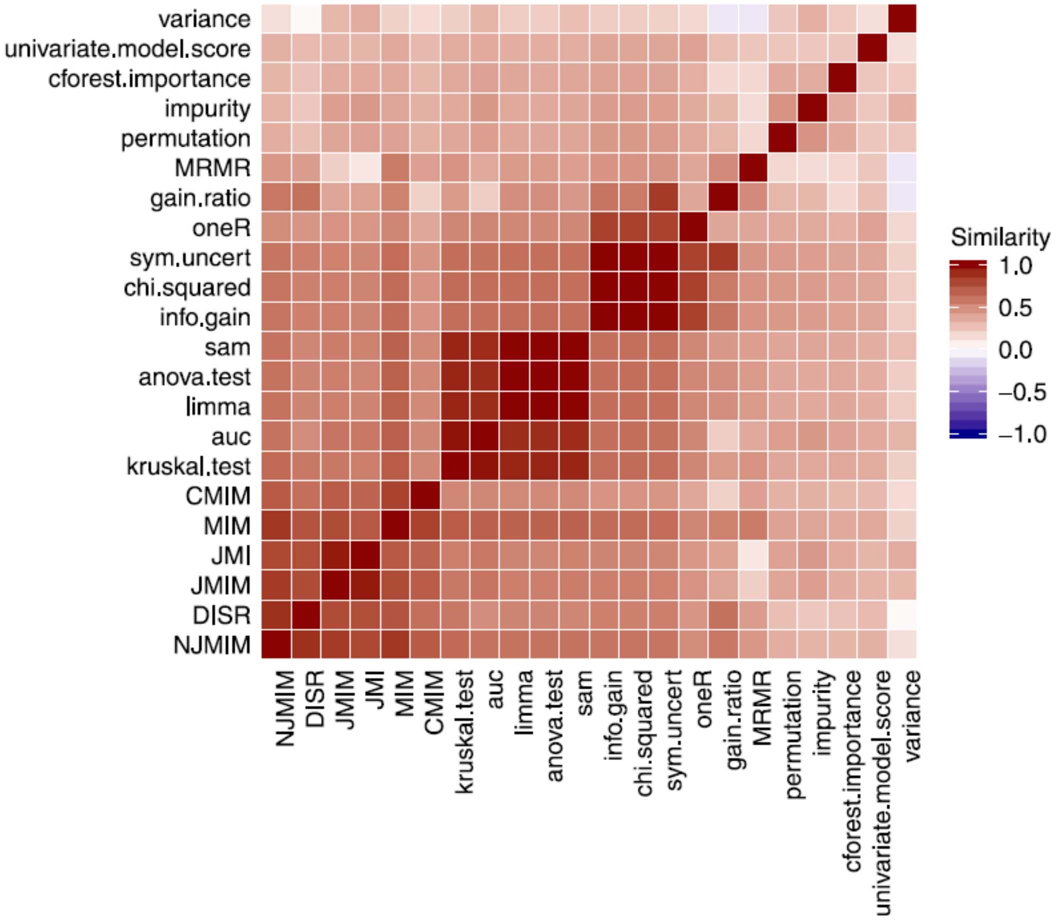


Figure 2.6: Rank correlations between pairs of filter selection methods. The figure is from the paper of Bommert A. et al. [36]

We can see that the chi-square (chi.squared above), ANOVA (anova.test) and MI (JMI) have relatively low similarity between them. This further supports our choice of the aforementioned feature selection methods. Since the methods do not have a high correlation between them, they approach the task from a different angle and will potentially have more interesting results to compare.

From the feature selection methods discussed, chi-square was the only one previously used with the TraNCE framework. One of the aims of the project was to investigate more feature selection methods, both wrapper and filter style, and evaluate their performance, focusing both on accuracy and computational cost. As discussed earlier, one of the main challenges of wrapper-style feature selections is the very costly computational time. As we will see later in Section 5.3.1, the feature space is very large, hindering the use of wrapper-type feature selections, and in extreme cases the use of filter-type methods.

We have identified the aforementioned problems, and will now discuss our approaches to them. We focus on how we can optimize UDFs that use feature selection while simultaneously improving the feature selection methods as well as downstream model quality. In the example GMB pipeline (2.5), the challenge of the large feature space still remains. The challenge of the large dimensionality of the feature space is discussed in Section 3. While there is no restriction in which external library is to be used to transform the Spark Dataframe outputted by TraNCE, in the thesis we will focus on using Python. A detailed discussion on the multi-class neural network and the one-vs-rest networks for classification can be found in Sections 5.4.1 and 5.4.2.

Section summary:

In this section, we discussed challenges associated with biomedical data such as the distributed processing of nested data. We then introduced TraNCE at a high-level, a nested query compilation framework that overcomes such challenges, and then introduced UDFs. We then presented different feature selection methods we will use in this thesis and how the curse of dimensionality is a big issue when it comes to biomedical data. We also noted how significant UDFs are as they support external libraries for downstream analysis, including processes such as feature selection. Moreover, due to the black-box nature of UDFs, we also noted how challenging it is to optimize them. As discussed, currently TraNCE does not support UDFs. In the next section, we discuss how we extended TraNCE to support UDFs at a high-level, and how we integrated feature selection methods with UDFs. After our extensions, the feature selection optimization methods could be used not only in external libraries but within TraNCE as well.

3 Extended TraNCE

In this section, we describe how we extended the TraNCE framework. We outline in detail the integration of UDFs in the framework and discuss some of the UDFs that were implemented. As mentioned in Section 2.5, currently TraNCE does not support UDFs. One of the aims of the thesis was to extend NRC and the TraNCE framework to work with UDFs and integrate them into the framework. We recall the architecture of TraNCE's execution of a program prior to our work in Figure 2.2. In the figure below, we present the transformation steps of the same program execution *after* we extend TraNCE with UDFs.

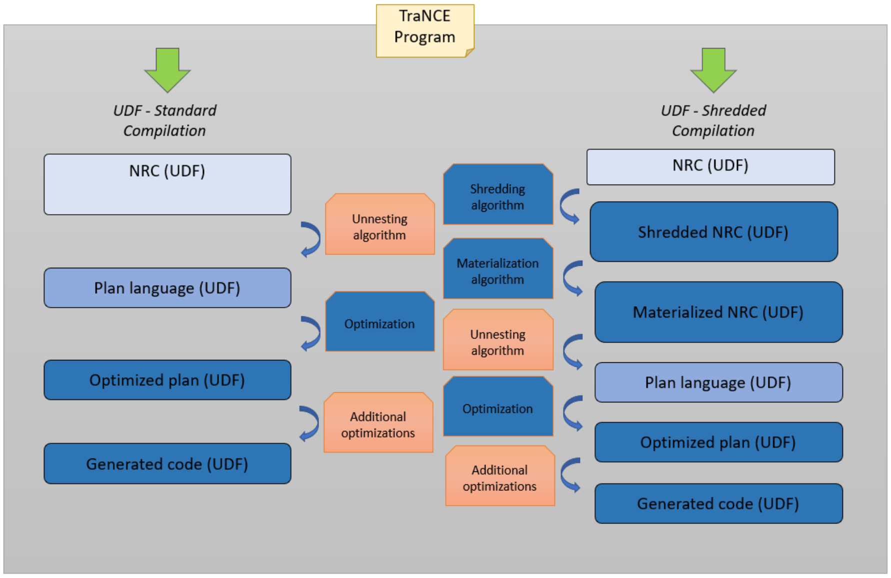


Figure 3.1: Transformation steps from raw input program to generated code using the UDF implementation. The darker shaded boxes denote the transformations where we had to change the implementation of TraNCE.

We note that for both standard and shredded compilations, UDF-handling happens at all the transformation steps, denoted by (UDF). The darker shaded boxes denote stages at which significant additions in the framework had to be made. Additions included component extensions (algorithmic extensions), changes in the optimizer, and so on. In the subsections that follow, we break down how we integrated UDFs in the TraNCE framework and how we optimized them with a hint optimization.

3.1 UDF Construct and Compilation

The first task was to define the input syntax for a user to define a UDF. We add an additional expression to the NRC source language presented in 2.4.3. Specifically, $udf^{name}_{params}(e)$ where *name* corresponds to the function name as defined by the user, *params* to the list of arguments that the user can pass to the function, and e to an NRC expression that acts as input to the UDF. Note that e must conform to the type system of the source language. The output type is an implicit parameter that is provided by the user and also complies with the same type system of the input.

For the Scala implementation in the TraNCE source code, we first define a UDF trait that summarized a generic UDF expression in the TraNCE source language. The source code is as follows:

```
trait Udf {

    // the name of the UDF
    def name: String

    // the input expression to the UDF
    def in: Expr

    // the output type of the UDF
    def tp: Type

    // additional parameters passed to the UDF
    def params: List[String]

}
```

We then extend various `Expression` types that are defined in the TraNCE framework and ensure strong typing in the language. For example, `NumericUDF` extended `NumericExpr`, `PrimitiveUDF` extended `PrimitiveExpr` and so on. The `NumericUDF` for example outputs a Numeric type result, `PrimitiveUDF` a Primitive type result etc. We then move through the list of transformations displayed in Figure 2.2, handling the relevant transformations on a UDF throughout the course of the compilation pipeline. For the standard compilation route, the handling of a UDF is straightforward: the UDF is passed through at each transformation step. At the code generation process, the UDF is translated into the respective function call. The definition of any UDFs used in the NRC program must exist in the UDF registry which provides the output type of the UDF and a corresponding `.udf` file containing the UDF source code. The code generation process keeps track of the UDFs that are referenced in the input program and their definition is encapsulated in the generated Spark application. We now provide two simple examples of UDFs for clarity, one of primitive type and one of bag type. First, consider a function `exampleudf` that appends a string to a specific column in the samples dataset. The user will first register the UDF for use in the framework, defining the output type as String and providing the definition in a file `exampleudf.udf`. This is a file containing a function defined in the Spark/Scala API as follows:

```
def exampleudf(input: Column, add: String): Column = concat(x, lit(add))
```

Once registered, the function can then be used in a TraNCE program. For example, a user can append the string “ID” to each of the `bcr_patient_uuid` attributes in the samples dataset:

```
AppendData <= For s in samples union
                {( sid := exampleudf(s.bcr_patient_uuid, "ID") )}
```

Recall that the unnesting transformation translates source NRC to a form (plan language, Spark code) that is designed for execution in bulk. This means the definition of the UDF will follow the bulk execution strategy of the TraNCE plan language. However, the user applies the function in NRC as a String ⇒ String transformation. This UDF NRC expression will be passed through the stages of the compilation route and eventually, the contents of `example.udf` and the corresponding function call as defined by the execution plan are written out to the Spark/Scala application in code generation. Now consider another UDF, `occurIdentity`, that performs the identity function on the occurrences dataset and thus returns a bag type. This UDF is registered to the system, providing the output type as the type of occurrences as well as the corresponding file definition in Spark/Scala. This is a file, `occurIdentity.udf`, that exists in the registry as:

```
def occurIdentity(input: Dataset[Occurrence]): Dataset[Occurrence] =
    input
OccurIdent <= occurIdentity(occurrences)
```

This function call merely gets passed through the whole standard compilation route, eventually generating the necessary code to perform the identity function call in Spark. However, when the users choose to run the above program using the shredded compilation route, the transformation becomes more involved.

3.2 Shredding UDFs

The most challenging task was to handle the UDF case in the shredding transformation step. Recall that there are two components to address in the shredded compilation route: the shredding and materialization algorithms. The shredding algorithm takes a source NRC and produces a symbolic expression. This symbolic expression is then passed to the materialization algorithm to produce the set of flat expressions corresponding to the shredded query. So now, we need to understand what it means to shred a UDF. If the goal of query shredding is to produce a set of flat queries that operate on flat inputs, then the goal of shredding a UDF is to produce a set of UDFs with flat output types that operate on a set of flat inputs. The shredded UDF thus consists of a collection of UDFs that each has a flat output type. Moreover, each UDF within the shredded UDF collection will accept the set of shredded inputs as input. For each UDF U, the shredding algorithm will expect an expression U^{flat} and U^{Dict} of the appropriate type to insert in recursive calls. For primitive types, this is simple; the UDF is just a flat component of the primitive UDF with a corresponding empty dictionary type. For bag types, this is a bit more complicated. Since we cannot analyze the structure of U within the compiler, these functions need to be supplied by the user. Because the user can not provide symbolic expressions in the symbolic shredding phase, U^{flat} and U^{Dict} are left as stubs. This stub is provided as an extended dictionary type that takes the symbolic representation of the shredded input, any additional parameters specifically passed by the user, and the shredded output type:

```
trait ShredUdf {
    // the name of the UDF
    def name: String

    // the flat input expression to the UDF
    def flat: Expr

    // the dict input expression to the UDF
    def dict: DictExpr

    // the output type of the UDF
    def tp: Type

    // additional parameters passed to the UDF
    def params: List[String]

}
```

Thus, the stub is a dictionary type that is able to pass the additional metadata of a UDF through to the materialization phase. At the materialization stage, we use the shredded output type to produce the set of materialized UDFs of flat output type expected from the materialization algorithm. In materialization, functions that provide materialization of the top-level and the flat component of each level of U^{Dict} need to be provided by the user. We now present the output of the shredding transformation on the `occurIdentity` function. In this running example, the stub would be:

```
ShredUdf(

    def name: String = "occurIdentity"

    def flat: Expr = occurrences_Flat

    def dict: DictExpr = occurrences_Dict

    // shredded output type
    def tp: Type = dictTp(occurIdentity.tp)

    // additional parameters passed to the UDF
    def params: List[String] = Nil

)
```

This is then passed to the materialization phase. Since there is a top-level and two nested collections in the type of occurrences, the final, materialized, output of the shredding transformation will have three expressions. For the running example, the materialized dictionaries for each level would then be as follows:

```
// output type matches the flat type of Top_occurrences
occurIdentity_Top(Top_occurrences, occurrences_Dict_1,
   occurrences_Dict_2)

// output type matches the flat type of Dict_occurrences_1
occurIdentity_Dict_1(Top_occurrences, occurrences_Dict_1,
   occurrences_Dict_2)

// output type matches the flat type of Dict_occurrences_2
occurIdentity_Dict_2(Top_occurrences, occurrences_Dict_1,
   occurrences_Dict_2)
```

The UDF registry does not require any additional type information or a shredded UDF, since this is handled in the shredding transformation. However, to activate shredding support for a UDF the user must provide the set of shredded functions expected by the output type. This means that the system expects three files, one for each level of the output type, which follows the output of materialization. The function definitions in Spark/Scala for each UDF in the shredded UDF collection for `occurIdentity` are below. Recall that this function will return the identity, so the contents of the function just return the shredded input for that level:

```
def Top_occurIdentity(top: Dataset[OT], first: Dataset[OD1], second:
   Dataset[OD2]): Dataset[OT] = top

def Dict_occurIdentity_1(top: Dataset[OT], first: Dataset[OD1], second:
   Dataset[OD2]): Dataset[OD1] = first

def Dict_occurIdentity_2(top: Dataset[OT], first: Dataset[OD1], second:
   Dataset[OD2]): Dataset[OD2] = second
```

where `OT` is the case class corresponding to the flat type of `Top_occurrences`, `OD1` is the case class corresponding to the flat type of `Dict_occurrences_1`, and `OD2` is the case class corresponding to the flat type of `Dict_occurrences_2`. After the shredding transformation, the compilation route proceeds as in the standard route, following through to code generation where the shredded definitions and corresponding function calls are applied in the Spark application.

3.3 External Types

In this section, we describe the extensions required to support externally-typed UDFs. We have so far only talked about types that are native to NRC and therefore translated into the corresponding Scala types. We call these internally-typed UDFs. However, as we have discussed in the background section, UDFs are commonly defined in languages that are more common to data science workflows, such as Python.

Recall the GMB classification analysis outlined in Figure 2.5. The lower portion of the pipeline is done completely in Python, and moving from a Spark Dataset TraNCE output type to a Pandas Dataframe requires *context switching*. Context switching is a feature provided in notebook frameworks that allows passing data representations between code interpreters - such as Scala to Python. To support this we allow users to register a UDF with an external type, which is treated as a catch-all for types outside of TraNCE. Each programming-language specific external type is provided as an extension as `ExternalType` within the TraNCE framework, for example `PythonType extends ExternalType`. We also provide native internal-to-external UDFs that allow a user to cast an external type back to an internal-type, in order to use the output of the function in any downstream NRC expression.

An externally-typed function can be added to the UDF registry in the same way as an internally-typed UDF. When an externally-typed UDF is registered, the function definition (in the .udf file) is provided in the desired language, such as Python. We extend the code generation process to keep track of the externally-typed UDFs that are referenced in the input query. Since the code generation process targets Spark/Scala directly, we need to ensure that an externally-typed UDF is not called within the main body of the generated application. We do need to ensure that the input types to the internally-typed UDF are accessible to the other contexts. We use the context switching capabilities of Apache Zeppelin notebooks to register the materialized input of the internally-typed UDF as a temporary table using `createOrReplaceTempView`, which creates a multi-context view of a Spark/Scala Dataset.

This means that if externally-typed UDFs are used, the user must compile them to a Zeppelin notebook. All internally-typed UDFs are written out to paragraphs in the Zeppelin notebook - denoted with the interpreter associated to their specific type, i.e. `%spark.pyspark`. The function is called under the definition of the function in the interpreter-specific paragraph. The user is responsible for accessing the temporary views of the appropriate inputs within their function definition.

We will now walk through an example of an externally-typed UDF using the GMB classification analysis as an example, summarized in Figure 2.5. The following is the TraNCE program associated to this analysis, where `Output` returns the training accuracy of a model:

```
GMB <= for g in genemap union
               {(gene:= g.g_gene_name, burdens :=
                 (for o in occurrences union
                   for s in clinical union
                    if (o.donorId = s.sample) then
                      for t in o.transcript_consequences union
                        if (g.g_gene_id = t.gene_id) then
                            {(sid := o.donorId,
                            lbl  := s.tumor_tissue_site,
                            burden := if (t.impact = "HIGH") then 0.80
                                                  else if (t.impact =
                                                      "MODERATE") then 0.50
                                                  else if (t.impact =
                                                      "LOW") then 0.30
                                                  else 0.01)}).sumBy({sid,
                                                      lbl}, {burden}))};

Output <= trainUDF(GMB, {sid, lbl, _1,burden, ANOVA})
```

In the above program, a user has defined a single UDF for the whole training portion of the pipeline, `trainUDF`, i.e. the external library part of 2.5. The output from the programs is passed as feature vector from the NRC language to the externally-typed `trainUDF`. The final output of the UDF is the training accuracy of the model. The testing accuracy of the model is outputted by a different UDF that is defined for testing. In order to run this program, the user registers `trainUDF` with the UDF-registry denoting it as `PythonType` and providing the function definition in `trainUDF.udf`. The definition of a UDF for the shredded compilation route is shown below:

```
def trainUDF(top, first, pivot_id, lbl, gene_id, score_metric, feature_method
    ):

    # load necessary imports
    import
    ...

    # access inputs from TraNCE that are registered as temporary views
    data = sqlContext.table(first)
    data.printSchema
    df = data.toPandas()
    ...
    # pivot dataframe
    df.pivot(index = [pivot_id, lbl], columns = gene_id, values =
        score_metric).fillna(0.0)
    ...
    # feature selection
    topk_features = topKFeaturesANOVA(X, df, ifPlot = False, topK = 200) if
        feature_method == "ANOVA" else (topKFeaturesMutual(X, df, ifPlot =
        False, topK = 200) if feature_method == "MI" else
        (...)
        ...
        )
    # fit model, get accuracy
    history = model.fit(X_train, y_train, validation_split = 0.30, epochs=
        10)
    acc.append(history.history[’accuracy’])
    ...
    # return accuracy
    return accuracy
```

In the example above, the top and first parameters come from the output of the NRC program and represent the top and dictionary type outputs of the shredded compilation: `GMB_top` and `GMB_burdens_Dict` (Section 2.4.5). These are strings representing the names of the inputs, since they have been registered as a temporary view, and then accessed with `sqlContext.table` within the Python paragraph. The user defines a list of parameters that are passed as values in NRC. The pivot_id, lbl, gene_id, and score_metric parameters ask the user to input the parameters they want for the pivot function, [47]. The pivot function is a Pandas function that groups by (over pivot_id and lbl) the column (gene_id) given and sums over the values (score_metric). Finally, the feature_method parameter asks the user to input the feature selection method they wish to perform. They can choose from the list of: {chisq, MI, ANOVA, MultiSURF, RFE}, otherwise, their input is not valid. These aspects are specific to the `trainUDF function` - the core of each UDF function depends on the task.

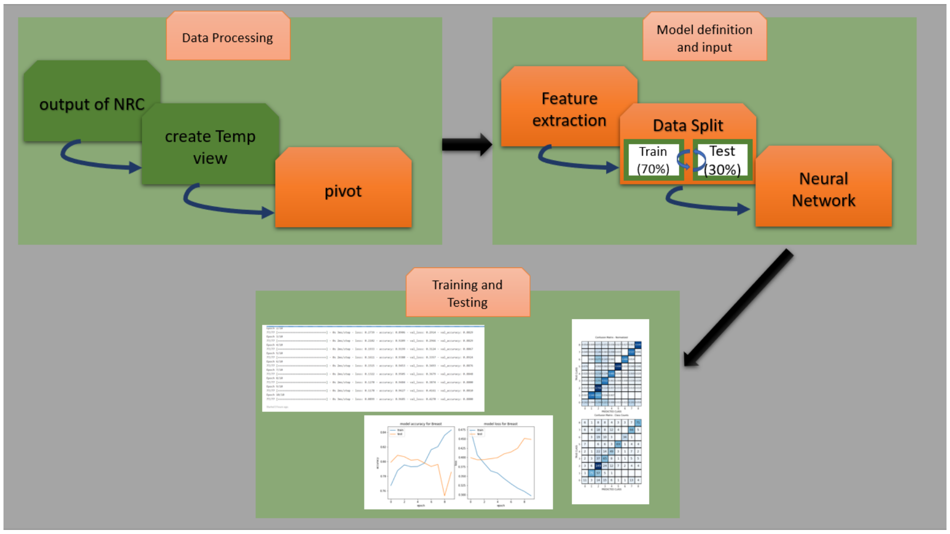


Figure 3.2: Outline of the steps defined in the UDFs. Data is processed initially in Spark (darker shaded boxes) and then in python. A neural network is then defined and trained on the extracted features.

To summarize, the example UDF above is provided as the truncated view of a more complicated UDF function. Figure 3.2 summarizes the main steps of the classification-based UDFs that were the focus of this thesis. As we will see in the next Sections, 5.3.2 and 5.4.1, different tasks involve different datasets and methods of feature selection. For the thesis, there are two UDFs defined for each of the different tasks discussed in the Experiments Section, Section 5. Namely, there are two UDFs for the binary classification task, two UDFs for the multi-class classification tasks, and two UDFs for the one-vs-rest classification tasks. For the full list of the defined UDFs, we refer the reader to the Github repository, [48].

3.4 UDF Optimization Hints

The black-box nature of UDFs means that program analysis would need to be done in order to identify what types of operations were capable of being optimized. We remove the need for program analysis by allowing the user to specify a hint within the application of a UDF within the NRC of a TraNCE program.

The UDF definition from Section 2.5 is extended with a hint parameter which takes an input string and additional parameters specific to the hint, if relevant. We update the UDF definition 2.5 to support this additional information as an optional input:

```
trait Udf {

    // the name of the UDF
    def name: String

    // the input expression to the UDF
    def in: Expr

    // the output type of the UDF
    def tp: Type

    // additional parameters passed to the UDF
    def params: List[String]

    // optional hint info
    def hint: Option[(String, Option[Double], Option[Double])

}
```

Users can then define the hint as an additional parameter inside of a UDF that exists in the registry:

```
... <= trainUDF(GMB, {sid, lbl, _1,burden, ANOVA}, udfhint)
```

Note that though the hint is supplied within the UDF function call in the NRC program, the hint information is captured at the level of the optimizer. In the optimizer, the hint will be translated into an additional node in the query plan that states where the appropriate filter will be applied. Thus the hint information is only used internally - it will not impact the code generation of the downstream UDF function call specified by the user. We extend the optimizer with a component that can detect a UDF node within the query plan and analyze the hint parameter. Query optimization is also a significant topic for optimizing runtimes, but due to time constraints was not investigated in depth. We note, however, that we present an optimization specific to multi-omics queries in the appendix in Section B.

Section summary:

In this section, we described how we extended the TraNCE framework and how we made changes that were UDF-specific for the majority of the steps in transforming the NRC program into generated code. We also note that we have now enabled analysis that was previously done in external libraries, to be done in the TraNCE framework via the UDFs. Finally, we introduced a hint optimization. Having introduced the hint, in the next section we discuss how we use it to optimize UDFs, focusing on feature selection filters.

4 UDFs with Feature Selection Filters

In this section, we describe a use case for UDFs and the related hint optimizations specific to feature selection in ML tasks. We first discuss in detail how we defined the filters and then how we implemented them in TraNCE. We then describe hints related to the filters.

4.1 Feature-selection Based Filters Introduction

As discussed earlier in Section 2.6, the feature space when dealing with biomedical data is large. The runtime of wrapper-style feature selection methods, such as RFE, is dramatically increased in such big feature spaces, making their use virtually impossible. For this reason, we decided to investigate how pushing filters upstream into NRC affects first the runtime of such filters, and second, affects the accuracy-related performance of the models. The runtimes of these filters can be found in the Experiments Section, in Section 5. We present below an explanation of the two filter-pushing methods that we provide in TraNCE, which we call *feature-selection based filters*. These are filters based on chi-square and mutual information feature selection methods. Correlation is an example of how a full calculation, i.e. calculating the Pearson correlation coefficient precisely without estimations, can be pushed into NRC while chi-square is provided as an example of how partial calculations can also be pushed into NRC. The general idea is that we can push these lightweight calculations into the NRC query in order to remove features that are likely to get thrown out from any feature selection method. This will speed up wrapper-style methods and help localize filtering and embedding methods to increase the quality of their feature sets.

4.1.1 Correlation Filter

The mutual information filter was discussed in Section 2.6. For a predictive model, we want to reduce the feature space and end up with features that maximize the mutual information between the selected features - a subset of the initial features - and the target variable. However, maximizing such quantity is an NP-hard optimization problem, due to the exponential number of possible combinations of the features. Due to the complicated nature of the filter, we decided to use an approximation of it and hence considered a correlation calculation. The correlation calculation is an ideal candidate because the full calculation can be pushed upstream without any heuristics. The correlation coefficient, ρ is defined as:

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

The above calculation shows how ρ is calculated between a feature X, and the target variable Y. $\bar{x}$ is the mean value of the feature simply calculated from the sum of the x_i values divided by the total number of values, n. The above calculation is then repeated for all features, and so in the end we have a list of correlations between all features and the target variable.

It was expected that the most important features will be the ones with a relatively large correlation (either positive or negative). On the one extreme, if we supposedly have two variables, X and Y (where X is the predictor variable and Y the predictand variable) and they are independent (implying $\rho = 0$), then knowing X does not give any information on Y, so we don't want X as our predictor. On the other extreme, if knowing X we can determine Y (so X is a deterministic function of Y), then we want to have X as our predictor to reduce the uncertainty of Y and make a good prediction. Correlation values fall somewhere in between, -1 and 1, $-1 \leq \rho \leq +1$, and so we want to keep the features that have values close to the two extremes, -1 and +1.

Finally, we need to define a threshold for deciding which features we want to keep. As an example, we could discard all features with $|\rho| < 0.1$. The features then with $|\rho| \geq 0.1$ are used as the input to the downstream analysis, meaning that they are still going to be further filtered by one of the feature selection methods discussed in Section 2.6. Hence, by defining the correlation filter, we cut down the large initial feature space, into one where the runtime of various feature selection methods is greatly reduced, as we will see in Section 5.3.

4.1.2 Chi-square Filter

Details about the chi-square feature selection can be found in Section 2.6. Due to the presence of hypothesis testing and the use of p-values, it was not possible to push the exact chi-square filter calculation upstream. For this reason, the filter we define does not include any hypothesis testing but involves the calculation of a variation of χ^2 value, that we call $\tilde{\chi}^2$ and define below:

$$\tilde{\chi}^2 = \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

Thus, the chi-square calculation is an example of how partial calculations can be used to push filters upstream into NRC. In the above equation, $\tilde{\chi}^2$ is calculated for all features, and so here we suppose that there are n observations for each of the features. O_k denotes the observed value of the feature and E_k represents the expected value. To give a simple example of how this is calculated, consider Table 1 below, where for simplicity we suppose that the target variable, Y is *binarized.* Binarization is the process of making a multi-label variable into one with labels involving just 0 and 1. For example, we can binarize a variable Y by creating columns for each different label of Y, and denoting by 1 for the value of Y we are interested in and 0 for everything else. Currently, the filter implementation does not support binarization which was manually done on occasions, but the future plan is to implement it along with the filters. n denotes the total number of unique values of the target variable, Y.

Table 1: $\tilde{\chi}^2$ **calculation example**

Observation	**Y**	**Feature_1**	**...**	**Feature_n**
1	0	4.6	...	1.0
2	1	4.2	...	1.4
3	1	4.5	...	1.6
4	1	5.0	...	2.0
5	0	5.3	...	2.5
6	0	4.0	...	0.9

Consider Feature_1. We perform a dot product between the feature and target. For Y = 0, we have a sum of 13.9 and for Y = 1 a sum of 13.7. We next calculate the class frequency, so P(Y=0) = 0.5 (as half of the observations have a 0 label) and P(Y=1) = 0.5 as well. We then calculate the sum of all feature values, so here count = 13.9 + 13.7 = 27.6. We now take the dot product to calculate the expected matrices. So for Y = 0, the observed value of Feature_1 is P(Y=0)*count = 0.5 * 27.6 = 13.8, and for Y=1, P(Y=1)*count = 0.5 * 27.6 = 13.8. We finally calculate the $\tilde{\chi}^2$ value, $\tilde{\chi}^2 = \frac{(13.9-13.8)^2}{13.8} + \frac{(13.7-13.8)^2}{13.8} = 0.001449$. The same procedure is then repeated for all features. For the original chi-square feature selection, this value is used for hypothesis testing. For the purposes of the local filter that we pushed, we sort these values. A high $\tilde{\chi}^2$ value indicates that the hypothesis of independence is incorrect, and hence the higher the $\tilde{\chi}^2$ value the more the feature is dependent on the response variable, and so can be selected for model training. Hence, as mentioned in the previous section, we define a threshold and keep features with scores exceeding that threshold, where the threshold can be adjusted depending on the desired number of features.

4.1.3 Filter Implementation

Since filters were pushed upstream into the NRC program, each implementation was in Spark/Scala and thus did not involve the use of any external libraries. Given that we focused on tasks that integrated datasets to produce light-weight feature vectors, an important advantage of pushing the filter upstream, i.e. into the TraNCE framework, was that we could reduce the data size as much as possible before it was passed to an external library, where we lose the ability to optimize ourselves. Another benefit was that we could leverage the distributed nature of the data within TraNCE, which was not guaranteed by an external environment. We now discuss implementation details and note specifics to the shredded compilation route when necessary.

When filters are pushed upstream from an external environment into a distributed environment, it is important to ensure parallelization of the calculation due to the often large number of features present. Because of the need to parallelize the calculation, many partitions are working in parallel. This means that in order to perform the calculation without moving data around, all the values associated with a specific feature must exist on the same partition. We guarantee this by the structure of our input program. The top-level contains the feature names and the first-level contains the (sample, label, value) tuples where samples is the row identifier, label is the predictor label and value is the feature value in the feature matrix. For the shredding procedure, the lower-level dictionary is a single table partitioned by feature name - meaning that all the values for each feature are guaranteed to be within the same partition. This allows us to apply calculations in parallel across data distributed by feature name, thereby speeding up the calculation. The calculations mentioned above are applied locally at each partition, and the thresholds are also applied locally at each partition. There is a benefit in the use of thresholds since we avoid having to do a global sort across all features once those that survive the threshold are collected. If we were to use a method that required picking the top-valued features, then this requires globally sorting all the values and that would be less optimal. This is because sorting features locally will lead to only selecting features that are optimal based on the local maximum and not the whole feature space.

As a final step, we collect the surviving features into a local set and broadcast that set to each partition to apply as a filter in the first-level dictionary. Note that because partitions in distributed frameworks are immutable, filtering cannot happen locally within each partition because we need to manipulate the partition data as a separate transformation. However, since the surviving feature list is assumed to be relatively small, i.e. less than the input feature set and only the names of said features, then the broadcasting cost is amortized. We now discuss how filters are integrated into the TraNCE framework.

4.2 Feature-selection Based Hints

Hints are available for any UDF that performs downstream analysis that will filter out features based on a relationship to a predictor label. We make two filters available in the framework in order to reduce the feature space. The hint is provided to allow the user to choose between the two filters, chi-square (chisq) or correlation (corr). Feature-selection filters have an upper bound threshold and a lower bound threshold as an extra parameter. If thresholds are not supplied defaults are used. We use a default of 0.01 since if users do not input a threshold then we do not want to filter out any important features. If no hint is supplied, then no filter is applied.

If a feature-selection hint is specified in a UDF, a new filter-based UDF is defined as a node above the original UDF node, taking as input the input passed to the original UDF. This will signal to the code generation process to apply and generate the code associated with the local filter, with the user-defined thresholds. The original UDF node is then defined with the same definition as before, but instead takes the output of the filter-based UDF as input. The design choice to use the UDF extensions to apply UDF-based filters was made for two reasons. First, it minimizes the extensions required to support UDF optimizations. Second, this automatically allows users to add their own filters into the framework. Users can define their own filters by registering their filter definition and metadata to the UDF registry. For more information and example filters, see [50].

Two other optimizations, which we refer to as *partial filters*, can be applied to further speed up the advanced filtering methods. We provide an additional hint that can be supplied to a UDF as a way of performing an initial filtering of features depending on the threshold, particularly small thresholds, provided by the user. In that case, features that will be filtered out are very likely to be discarded by any of the feature selection methods and so do not affect the performance of the model. This is based on the assumption that feature-selection based filters focus on the strength of the dependency between a feature and a target variable. For more details on an experiment performed on this, we refer the reader to Section 5.4. These are available as additional filter hints called `chisq+` and `corr+`.

We also explored pushing two other types of filter, *source filter* and *value filter*. The source filter is pushed into the input data source before the feature values are aggregated (default is 0.01) while the value filter is pushed after the feature values are aggregated. Both filters can be particularly useful when the data size is large or cluster resources are low. This pushing requires a bit more involved analysis of the query plan, so it was only applied manually for the sake of exploration. In the current implementation, we allow both the source and value filter to be used independently of the feature-selection based filters.

Section summary:

In this section, we introduced the feature-selection based filters (correlation and chi-square) along with partial, source and value filters. These filters aim to reduce the feature space from the output of programs by using methods that will avoid where possible the loss of informative features. In the Experiments Section, we make use of the UDFs we introduced and the different filters to investigate the overhead the filters add to the runtime of the NRC program and the downstream feature selection. Also, we run experiments on two classification tasks (binary and multi-class) and investigate the performance of the models with and without the use of the filters. Models that do not use filters, and hence have as input the full initial feature space, are considered the *baselines*. We then investigate the features outputted by some of the best performing models by performing gene enrichment analysis.

5 Experiments

So far, we have introduced the TraNCE framework and feature selection methods. We have also defined feature-based filter methods and how to combine them with UDFs which act as a training and testing pipeline. We have also outlined how we extended TraNCE with UDFs and discussed an optimization hint. We now want to test the implementation of UDFs on biomedical data sources, mainly focusing on runtimes and model accuracy. Our baseline is the model where no filter is pushed.

In this section we outline the experiments, describe the experimental setup, and present and discuss the results. We first outline the experimental setup in Section 5.1, then in Section 5.2 we discuss an initial feature analysis by which we determined the optimum number of features to input to our neural network, and function optimization. We then discuss the binary classification task for the severity of prostate cancer in Section 5.3. In that section, we present first three different programs of calculating GMB for predicting the severity of prostate cancer and compare their runtimes and the model performance. In Section 5.4, we introduce the pan-cancer multi-class classification problem. Initially a multi-class analysis, Section 5.4.1, and finally we discuss a one-vs-rest approach for the same task, in Section 5.4.2. Finally, in Section 5.5 we discuss analysis on genes extracted from different models and present our findings.

We outline in the list below the aims of the experiments:

1. Define thresholds for the input number of features at which RFE runs at a reasonable time and decide where it is best (Scala or Python) in terms of computational costs to run data processing functions, such as pivot functions

2. Investigate the runtimes of models both for the binary and multi-class classification tasks, comparing the baseline case where UDF is used without any filter against models with filters pushed

3. Investigate the accuracy of models for the two tasks we considered. First, the binary

classification task of predicting the severity of prostate cancer, and second the multi-class classification problem. For each task, we want to see which omics approach, (single or multi-omics), which feature selection method, and which feature-selection based filter results in the best performing models.

4. Analyze the feature sets from the best performing models of the prostate and multi-class experiments using gene sets analysis tools to assess their biological relevance

The experimental results can be summarized as follows:

- Pushing filters from UDFs into the NRC program can provide up to x16.5 speed up for filter-based feature selection. This enables wrapper-based methods to run to completion, which were previously unable to perform at all.
- Pushing feature-selection filters adds little overhead to the total runtime of the NRC program.
- Pushing filters from UDFs benefits binary classification methods for single and multi-omics approaches, reporting up to 99.3% accuracy for predicting prostate cancer severity.
- Partial filters pushed from UDFs are useful for large datasets, restricted cluster sizes, and can increase model performance for multi-classification problems.

5.1 Experimental Setup

For the experiments we discuss, we use the `occurrences`, `gene_expression`, `clinical`, `samples` and `genemap` data sources, introduced in section 2.2. All the data sources except `genemap` come from the TCGA dataset, while `genemap` comes from GRCh37. We also note that all the experiments were run with the shredded compilation framework and not the standard one. We recall the TMB calculation from Section 2.1. The experiments in this section focus on the sub-calculation of TMB, GMB, and different approaches for calculating GMB scores. The neural network (NN) that is used as our classification model is a fully connected, feed-forward NN. In the case of the binary classification tasks, the NN is made up of three dense layers. The first two dense layers have a LeakyReLU activation function with $alpha = 0.05$ and a dropout layer with the dropout rate being 0.15. The third and final output layer has a sigmoid activation function. The sigmoid activation function was used since we were performing binary classification and its output can be interpreted as a probability since it outputs values between 0 and 1. The model is trained using binary cross-entropy as the loss function and Adam optimizer for the gradient descent optimizer. For the multi-class classification tasks, the first two dense layers also have LeakyReLU activation functions, with $alpha = 0.05$, however, the first dense layer has a dropout rate of 0.30 while the second dense layer has a dropout rate of 0.20. The final output layer has a softmax activation function (due to the presence of multiple target labels) and categorical cross-entropy for the loss function. The gradient descent optimizer is the Adam optimizer. For all experiments, 70% of the data was used for training, and 30% for testing. The training set was further split into 70% training and 30% validation sets. All experiments were performed on a single node cluster with Spark 3.1.2 and Scala 2.12, one worker, 8 cores, and 100 G memory. The datasets we used for each experiment are discussed in the experimental setup of each experiment.

The Mutual Information (MI), chi-square, ANOVA, and RFE feature selection methods were implemented using python's package, `sklearn.feature_selection` [46]. For RFE, we used a random forest classifier as the estimator with a step size of 10. The estimator is a supervised learning model with a `fit` method that ranks features based on *feature importance*. Feature importance refers to techniques that assign a score to input features based on how useful they are at predicting a target variable. The step size corresponds to the number of features that are removed at each iteration. MultiSURF was implemented using the `skrebate` package, [52]. Experiments will include runtimes, the program we ran to calculate GMB, and accuracy results.

5.2 Exploration

The aims of this section were to define thresholds for the input number of features at which RFE could run at a reasonable time and investigate the pivoting function.

The number of features to be used as the input of predictor models (from here on referenced as topk_features) relative to the total number of observations is a very important and challenging question for machine learning tasks. Following the one in ten rule, detailed in [43] and the general consensus in the computer science world, it was decided that at most 10% of the total number of features were to be used as an input to the model, as otherwise the model would overfit [17]. To investigate this, we present the results of the calculation of TMB using GMB as shown in the program below and plot the training and validation accuracies of the neural network ran using a big number of features, ~ 20000:

```
GMB <=
    for g in genemap union
       {(gene:= g.g_gene_name, burdens :=
       (for o in occurrences union
           for s in clinical union
               if (o.donorId = s.bcr_patient_uuid) then
                  for t in o.transcript_consequences union
                    if (g.g_gene_id = t.gene_id) then
                       {(sid := o.donorId,
                        lbl := if (s.gleason_pattern_primary = 2) then 0
                          else if (s.gleason_pattern_primary = 3) then 0
                          else if (s.gleason_pattern_primary = 4) then 1
                          else if (s.gleason_pattern_primary = 5) then 1
                          else -1,
                         burden := if (t.impact = "HIGH") then 0.80
                                  else if (t.impact = "MODERATE") then
                                     0.50
                                  else if (t.impact = "LOW") then 0.30
                                  else 0.01
                        )}
             ).sumBy({sid, lbl}, {burden})
           )}
```

We note that the training and validation accuracies shown below, come from the training of the neural network from the experiment of the binary classification tasks discussed in Section, 5.3.1. Figure 5.1 uses chi-square as the feature selection method and extracts 20 000 features from the output of the program above (which was around 55 000), while Figure 5.2 uses ANOVA as the feature selector again with an output of 20 000.

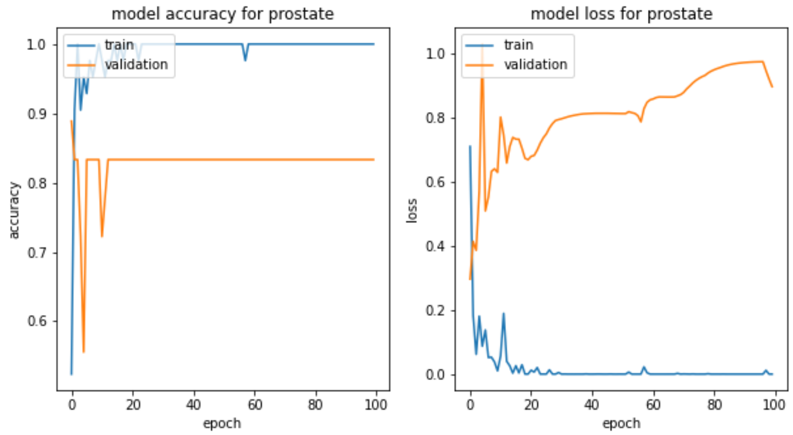


Figure 5.1: Training and validation error of the model using chi-square as feature selector method. The model uses 20000 features as its input coming from chi-square. The graph clearly displays overfitting problems.

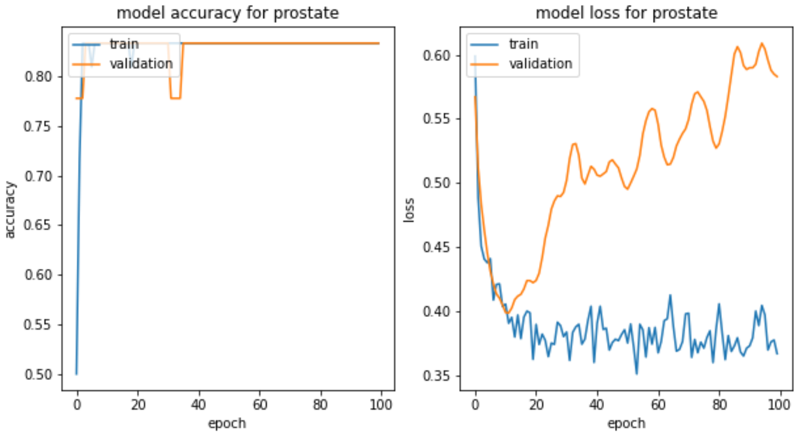


Figure 5.2: Training and validation error of the model using ANOVA as feature selector method. The model uses 20000 features as its input coming from the ANOVA filter. The graph clearly displays overfitting problems.

In both graphs, we can clearly see an overfitting pattern. The model loss for validation increases while the loss for the training decreases as the number of epochs increases. Therefore, also following the 10% rule, less than 5000 features were considered for the topk_features variable. To explore this further, cross-validation was run across 5 folds using both chi-square and ANOVA. These two feature selection methods were chosen for time efficiency since they only took one second to run. The cross-validation was run for 100 epochs and we present the results in the table below. The columns denote the number of features used, while the percentages are the testing accuracies.

Table 2: **Severity of prostate - topk_features cross validation**

Method	**5000**	**4000**	**3000**	**2000**	**1000**	**200**	**100**
chi-square	80%	80%	75%	80%	78%	**85%**	70%
ANOVA	80%	82%	82%	81%	84%	**88%**	72%

As we can see from the results above, topk_features = 200 outperforms all other number of features for both chi-square and ANOVA. It was therefore decided to use topk_features = 200 for this experiment and for the ones that follow.

Moving on to the optimization part, we thought it was worth considering where to execute the pivoting function since it was used for all of our experiments. There were two options; either in python using Pandas, or within the TraNCE framework (and so in Scala). We hence investigated when the runtime of the pivoting was "reasonable" [9] in Scala by trial and error:

- with 500 000 rows runtime was > 5 minutes
- with 400 000 rows runtime was > 5 minutes
- with 300 000 rows runtime was > 5 minutes
- with 200 000 rows runtime was > 5 minutes
- with 100 000 rows runtime was > 5 minutes

We note that initially, the number of rows outputted by the programs was in the range of 400 000-600 000 rows. We also note that after 5 minutes passed, we ended the execution of the function.

---

[9] Here the time which we accepted as reasonable for the pivoting operation was 5 minutes

On the other hand, pivoting in pandas, took 21 seconds for any number of rows input with very slight differences in time (in the range of 1-2 seconds). The fact that it took at least x15 more for the pivoting operation to execute in Scala affirms the choice of scientists to often choose Python to define their UDFs since pivoting is a crucial step in the data handling process. Having performed an initial exploration, we now continue to the experiments.

5.3 Binary Classification: Prostate Cancer Severity

5.3.1 Single-omics - Mutation Impact Burden

The aims of this experiment included the investigation of the overhead the pushed feature-selection based filter adds to the total runtime of the program for the binary classification task using a single-omics approach. We also aimed to investigate the test accuracy of predicting the severity of prostate using a single-omics approach (`occurrences`), comparing our results against an initial model that does not use a feature-selection based filters. The `genemap`, `occurrences` and `clinical` data source were used and the size of the files were 1.2GB, 4.67 GB and 369 KB respectively. For the mutation impact burden calculation, we used the TCGA dataset with the Prad somatic mutations which are prostate-specific. Our aim was to predict the severity of prostate cancer from a list of patients diagnosed with prostate cancer. The Gleason scoring system is one of the most common prostate cancer grading systems used. Pathologists monitor the arrangement of cancer cells and assign a score for two different locations, primary and secondary - we focused on the primary location.

Following our discussion on wrapper-style feature methods from Section 2.6 and their long runtimes, it was of interest to come up with a threshold on how many features RFE could run at a reasonable time (maximum of 25 minutes). We present in Figure 5.3 below the runtimes of the 5 feature selection methods we considered. These runtimes come from the execution of the GMB program introduced in the previous Section, 5.2.

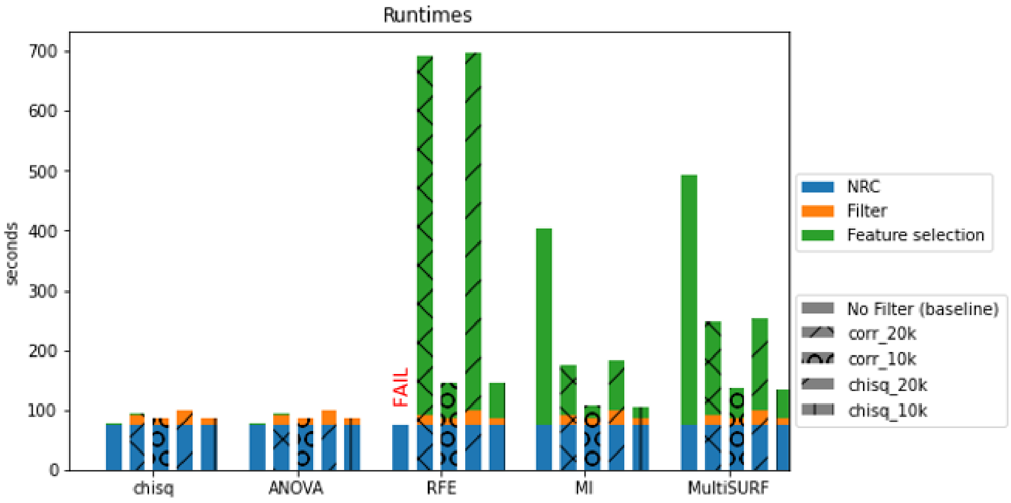


Figure 5.3: Stacked bar plot showing the runtimes of different feature selection methods of the calculation of the single-omics mutation impact burden calculation from the moment the program is executed until extracting the features.

In the Figure above, NRC (blue, bottom part of the bar) denotes the runtime of the program (which is common for all feature selection methods). Filter (orange, second part of the bar) denotes the runtime of the filter (either correlation or chi-square). The first stacked bar of each method does not include filter runtime as we used all the features outputted by the program. Feature selection denotes the runtime it took for the 5 methods respectively to run given the input. corr_20k denotes the correlation filter with 20 000 features as input, corr_10k with 10 000 features, and so on. The final execution time is calculated by adding the 3 runtimes, NRC + Filter + Feature selection. Moreover, for chi-square and ANOVA, the runtimes for the respective Feature selection is around 1 second, which is why they are not visible from the plot. We also note that RFE fails to run (the Zeppelin interpreter crushes) without any filter and is denoted on the plot with the label “FAIL”. We can see that the runtimes of the filters decrease as the threshold becomes more strict. For both corr_20k and chisq_20k the runtimes are much bigger than the corresponding ones from corr_10k and chisq_10k. We can see that RFE's runtime is the largest but decreases dramatically when the threshold is stricter (corr_20k compared to corr_10k). We can also see major differences in the runtimes between feature selection methods. Both chi-square and ANOVA have very small runtimes compared to MI and MultiSURF. For the full feature selection runtimes, since they are not visible clearly in the graph due to the big difference in runtimes across feature selections, we refer the reader to Section A. In summary, we see up to an x16.5 speed up for filter-based feature-selection and are able to run wrapper methods to completion which were previously unable to perform at all.

The program we used was the one from the previous Section. 5.2. As shown in the program, after joining the different data sources we labeled as 0 tuples with `gleason_pattern_primary` scores being 2 or 3 as they were considered low in the medical community, and scores of 4 or 5 were considered high.

Initially, we ran the results of the program using all the features (55000) outputted (and hence without using any of the feature-selection based filters discussed in Section 4.1). These are the results after 100 epochs, performing the experiment 100 times and taking the average from all the runs and by using the train and test split mentioned in the introduction of the Experiments Section. We also note following our previous discussion, that for this experiment and the ones that follow, topk_features was fixed to 200. Also, we will refer to models by the specific feature selection method used to generate the topk_features: chi-square, ANOVA, RFE, etc. We use subscripts to denote the number of features that are passed into the feature selection methods, for example, ANOVA_10000 means we are using the ANOVA model with 10000 features as input to the model.

Table 3: **Binary classification - Single omics Mutation impact, all features**

Method	**Testing Accuracy**
chi-square	60.1%
ANOVA	**78.3%**
MI	62.7 %

As we can see, ANOVA outperforms both chi-square and MI quite significantly - an 18.7% difference with chi-square and 15.6% with MI. Here, we note that there are no results for RFE as it took too long to run (more than 1 hour at which point the interpreter crashed) using all the 55000 features.

Results of the chi-square filter pushed upstream

In the table below, we present the results of the same analysis performed, but using the chi-square as a feature-selection based filter (discussed in 4.1). The output of the filter was a score, $\tilde{\chi}^2$ for each feature, where $\tilde{\chi}^2 \in \mathbb{R}^+$. Each time we wanted to output a different number of features, we defined a threshold and any features with $\tilde{\chi}^2$ less than the threshold was dropped. We expected that the bigger the score of the features, the more important they were for downstream analysis. We note that 20 000 features had $\tilde{\chi}^2 > 0.088$, 10 000 had $\tilde{\chi}^2 > 0.28$, 5000 had $\tilde{\chi}^2 > 0.86$, 2000 had $\tilde{\chi}^2 > 2.06$ and 1000 had $\tilde{\chi}^2 > 3.56$.

Table 4: **Binary classification - Single omics Mutation impact, chi-square feature-selection based filter**

Method	**20000**	**10000**	**5000**	**2000**	**1000**
chi-square	74.4%	74.1%	75.0%	74.9%	62.2%
ANOVA	83.8%	83.9%	70.9%	71.8%	77.6%
RFE	**84.8 %**	88%	79.5%	58.3%	58.3%
MI	59.3%	66.6%	52.3%	52.9%	55.2%

Pushing chi-square, not only benefited the results in the sense that RFE could now have been used (and in the end perform the best), but also improved the performance of all three other feature methods. In particular, ANOVA's performance improved from 78.3 % to 83.9% (using 10 000 features), chi-square increased from 60.1% to 75.0% (using 5000 features) and MI improved from 62.7% to 66.6 % (using 10 000 features). RFE was the best performing model achieving an accuracy of **84.8%**.

Results of the correlation filter pushed upstream

In the table below, we present the results of the experiment performed with the correlation feature-selection based filter. 20 000 features had low correlation with the target variable, 10 000 features had medium correlation and 5 000 had high correlation. Low correlation here denotes features having $|r| > 0.10$, medium denotes $|r| > 0.20$ and high $|r| > 0.90$. We note that we considered the absolute value of the correlation, r, since we were interested in both negative and positive correlation between features and target variables. If a feature had strong negative correlation, that still meant that it was useful in the prediction.

Table 5: **Binary classification - Single omics Mutation impact, correlation feature-selection based filter**

Method	**20000**	**10000**	**5000**
chi-square	75.2%	68.6%	62.1%
ANOVA	78.6%	78.1%	91.2%
RFE	90.2 %	93.1%	**94.0%**
MI	70.7%	70.3%	78.7%

Even though the correlation filter did not improve the accuracy outputted using the chi-square filter, it drastically improved the results for the other three methods. In particular, MI's accuracy improved from 66.6% to 78.7% (using 5 000 features). Even though the performance of MI was not the best one, it was expected that its performance was going to increase by pushing this filter. This is because, as discussed in Section 2.6, MI depends on the correlation between the features and the target variable, and hence by filtering out the features with lower correlation, MI was expected to select the better performing features when outputting the topk_features. Moreover, we saw a good improvement on both ANOVA_5000 and RFE_5000. ANOVA's performance increased from 83.9% to 91.2%, while RFE's performance increased to **94.0%**. Overall, compared to the baseline model, i.e. where all features were used with no filter, we see a big improvement in the model performance. Comparing with RFE_5000's accuracy which was 94.0%, we improved the best baseline model which was ANOVA with 78.3% by 15.7%.

5.3.2 Single-omics - Gene Expression

The aims of this experiment included the investigation of the test accuracy of predicting the severity of prostate cancer for the binary classification task using a single-omics approach (`gene_expression` this time), comparing our results against an initial model that does not use a feature-selection based filter as well as the previous results where we used the `occurrences` data source. The `genemap`, `gene_expression`, `clinical` and `samples` data source were used and the size of the files were 1.2GB, 167MB ,369 KB and 863 KB respectively. We used the same experimental setup as the one in the previous Section, 5.3.1 but the difference was the way we calculated the GMB. This time, we used the Fpkm score of genes. Fragments Per Kilobase of exon model per Million reads mapped, (Fpkm), are common units reported to estimate gene expression based on RNA sequential data. It is calculated from the number of reads mapped to a particular gene sequence. For more information, we refer the reader to [41]. For this experiment, we used the `gene_expression` data source and not `occurrences` as the Fpkm score information is associated with the gene expression of genes and not with their mutational occurrences. We present the program below noting that for this program, aside from using `gene_expression` instead of `occurrences`, we also had to join the genes on `samples` since there were no common attributes between `clinical` and `gene_expression`.

```
GMB <=
    for g in genemap union
        {(gene:= g.g_gene_name, fpkm :=
        (for e in expression union
            for c in clinical union
                for s in samples union
                  if (s.bcr_patient_uuid = c.bcr_patient_uuid) then
                    if (e.ge_aliquot = s.bcr_aliquot_uuid) then
                      if (g.g_gene_id = e.ge_gene_id) then
                         {(sid := e.ge_aliquot,
                           lbl := if (c.gleason_pattern_primary = 2) then 0
                            else if (c.gleason_pattern_primary = 3) then 0
                            else if (c.gleason_pattern_primary = 4) then 1
                            else if (c.gleason_pattern_primary = 5) then 1
                            else -1,
                           fpkm := e.ge_fpkm
                          )}
        ).sumBy({sid, lbl}, {fpkm})
      )}
```

The results shown below came from using all features outputted by the program above. The number of features outputted this time was 36 000 owing to the difference in the data sources, `gene_expression` data source and `occurrences` and the fact that we joined through a different data source (`samples`).

Table 6: **Binary classification - Single omics gene expression, all features**

Method	Testing Accuracy
chi-square	59.8%
ANOVA	**80.7%**
MI	67.8 %

Comparing the results above with the ones from table 3, we can see that we have very similar performance for chi-square and ANOVA, with chi-square's accuracy falling by 0.3 % and ANOVA's accuracy improving by 2.4 %. The MI model's performance however increased from 62.7% to 67.8%. This suggested that at least when using all the features, the Fpkm score was a better attribute to use for the prediction of prostate cancer severity compared to the impact score.

Results of the chi-square filter pushed upstream

In the table below, we present the results of the same experiment but using chi-square as the feature-selection based filter. We note that the thresholds defined for extracting different amounts of features were bigger, since the values of Fpkm are greater than impact values. 20 000 features had $\tilde{\chi}^2 > 280\ 000$, 10 000 features had $\tilde{\chi}^2 > 3.3 \times 10^6$, 5000 had $\tilde{\chi}^2 > 12 \times 10^6$, 2000 had $\tilde{\chi}^2 > 32 \times 10^6$ and 1000 had $\tilde{\chi}^2 > 52 \times 10^6$

Table 7: **Binary classification - Single omics gene expression, chi-square feature-selection based filter**

Method	**20000**	**10000**	**5000**	**2000**	**1000**
chi-square	60.3%	61.1%	61.3%	57.6%	58.9%
ANOVA	**76.8%**	74.5%	70.4%	64.3%	58.8%
RFE	56.7%	62.2%	62.2%	62.6%	54.8%
MI	62.2%	61.5%	61.5%	51.9%	54.6%

Comparing with the results from table 4, we see that the general trend is that the models perform worse. With the exception of MI_2000 and MI_5000, all other test accuracies are lower. The best performing model was ANOVA_20000. It is also worth mentioning that comparing these results with the ones from table 6 above where all the features were used, we also see that with the chi-square filter pushed the model accuracies fall.

Results of the correlation filter pushed upstream

In the table below, we present the results of the experiment performed with the correlation filter. Compared to the results we presented in table 5, we can see that we were able to distinguish with more granularity between the number of features. For more discussion on why this was the case, see Section 6. For this experiment, 20 000 features had $|r| > 0.10$, 10 000 features had $|r| > 0.15$, 5000 features had $|r| > 0.20$, 2000 features had $|r| > 0.40$ and 1000 features had $|r| > 0.95$.

Table 8: **Binary classification - Single omics gene expression, correlation feature-selection based filter**

Method	**20000**	**10000**	**5000**	**2000**	**1000**
chi-square	61.1%	62.2%	70.7%	70.6%	74.7%
ANOVA	83.3%	83.3%	83.4%	83.0%	83.2%
RFE	78.3%	76.3%	78.6%	87.4%	84.0%
MI	77.5%	83.4%	84.2%	87.2%	**87.6%**

We can see that MI_1000 outputs the best accuracy (with RFE_2000 features being very close), **87.6%**. This is a big improvement in comparison to the results of no filter pushed or chi-square pushed, as previously the best model had an accuracy of 80.7%. Moreover, comparing with the results from the previous experiment in table 5, we see that even though we don't match the performance of RFE_10000, which was 94.0%, the best performing model, MI_1000 takes as input 1000 features. Outputting 1000 features was not possible for the previous experiment using the same filter. We also note the overall improvement in the test accuracy with the use of the filter against the baseline model. MI_1000 outputted an accuracy of 87.6%, while the best performing baseline model, ANOVA, outputted an accuracy of 80.7%, an improvement of 6.9%.

At this point, we implemented MultiSURF, which was described in Section 2.6, using the same experimental setup. The testing accuracy was 71.7% accuracy and was obtained using all the features (hence without any feature-selection based filters). Because of the promising result, to further investigate MultiSURF, we tested it using the correlation filter. Even though we fixed topk_features = 200 for all other feature selection methods, we investigated what the optimal topk_features was for MultiSURF since it was not investigated before and as it was implemented using a different Python package. The table below displays the test accuracies of MultiSURF using a varying number of features as its input and the associated number of output features that MultiSURF outputted to be used for the neural network (30, 50, and 200):

Table 9: **Binary classification - Single omics gene expression, MultiSURF correlation feature-selection based filter**

MultiSURF Features Output	**55000**	**20000**	**10000**	**5000**	**2000**	**1000**
30	69.8%	62.0%	71.9 %	74.6 %	74.6 %	73.0 %
50	69.5%	51%	72.4%	73.1%	75.8%	72.5%
200	69.6%	73.6%	66.0 %	74.8%	**77.0%**	72.9%

We can see that the once more topk_features = 200 was the optimal choice, and hence topk_features = 200 was also used for MultiSURF. Comparing with the results from table 9, we can see that MultiSURF outperforms chi-square, with an accuracy of 77% compared to 74.7%.

5.3.3 Multi-omics - Integrated Impact and Gene Expression

The aims of this experiment included the investigation of the overhead the pushed filter added to the total runtime of the program for the binary classification task using a multi-omics approach. We also aimed to investigate the test accuracy of predicting the severity of prostate cancer using a multi-omics approach (combining `gene_expression` and `occurrences`), comparing our results against an initial model that did not use a feature-selection based filter and with the results of the previous two experiments. The `genemap`, `occurrences`, `gene_expression`, `samples`, and `clinical` data source were used where the size of the files were 1.2GB, 4.67GB, 167MB, 863KB and 369 KB respectively. For this experiment, we used the same experimental setup as with the previous experiments, however, we used a different approach to calculate GMB. Instead of using one data source to extract information about mutational burden scores, we combined two omics data sources. For a detailed discussion on the benefits of the multi-omics approach, we refer the reader to Section 2.1. We present in Figure 5.4 below the runtimes for this experiment. We note that the runtimes were very similar to the previous experiment in Section 5.3.2 where `gene_expression` was also used and so decided to present the runtimes of the multi-omics experiments only.

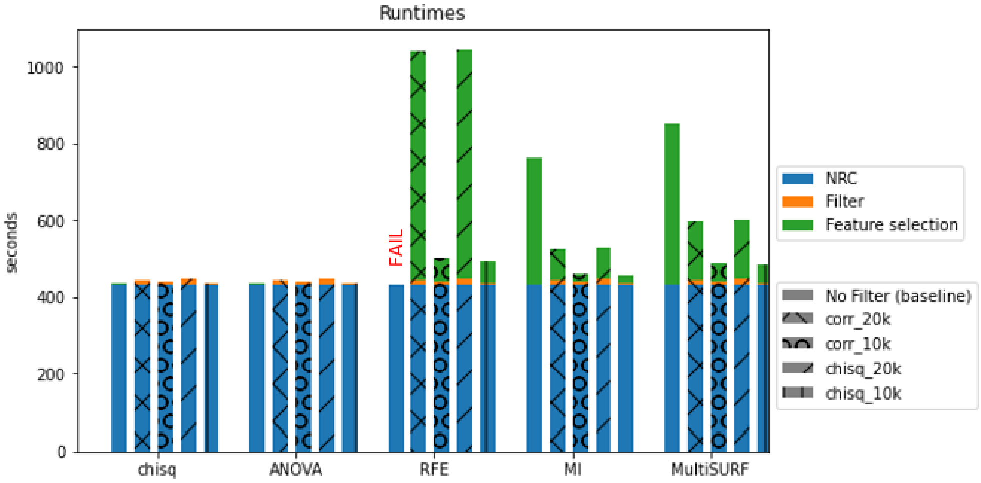


Figure 5.4: Stacked bar plot showing the runtimes of different feature selection methods of multi-omics calculation from the moment the program is executed until extracting the features.

For this experiment, we note that the runtime of the NRC query is much longer compared to the one in Figure 5.3 where the `occurrences` data source was used. This is because we integrated the `gene_expression` data source which is large. We can also see how small the runtime of the filters (orange, second bar) is compared to the total runtime, implying that the filter methods we use add minimal overhead to the total runtime.

We combine the `gene_expression` data source and the Fpkm attribute of the genes, with the `occurrences` data source and the impact attribute by multiplying them together. Due to the fact that the impact scores are relatively small (in the range roughly of 0 to 13), we expected that the importance of genes with somewhat high impact scores was going to stand out in this experiment since the impact scores were multiplied with the much larger Fpkm values (in the range of 0 to 1×10^6). We present the program below:

```
    mapExpression <=
            for s in samples union
              for e in expression union
```

```
             if (s.bcr_aliquot_uuid = e.ge_aliquot) then
              {(sid := s.bcr_patient_uuid, gene := e.ge_gene_id, fpkm
                  := e.ge_fpkm)};

impactGMB <=
         for g in genemap union
           {(gene_name := g.g_gene_name, gene_id:= g.g_gene_id, burdens
              :=
             (for o in occurrences union
              for s in clinical union
                if (o.donorId = s.bcr_patient_uuid) then
                  for t in o.transcript_consequences union
                    if (g.g_gene_id = t.gene_id) then
                      {(sid := o.donorId,
                        lbl := if (s.gleason_pattern_primary = 2) then 0
                         else if (s.gleason_pattern_primary = 3) then 0
                         else if (s.gleason_pattern_primary = 4) then 1
                         else if (s.gleason_pattern_primary = 5) then 1
                         else -1,
                        burden := if (t.impact = "HIGH") then 0.80
                                               else if (t.impact =
                                                   "MODERATE") then 0.50
                                               else if (t.impact =
                                                   "LOW") then 0.30
                                               else 0.01
                       )}
             ).sumBy({sid, lbl}, {burden})
           )};

GMB <=
         for g in impactGMB union
           {(gene_name := g.gene_name, gene_id := g.gene_id, burdens :=
             (for b in g.burdens union
              for e in mapExpression union
                if (b.sid = e.sid && g.gene_id = e.gene) then
                  {(sid := b.sid, lbl := b.lbl, burden :=
                      b.burden*e.fpkm)}).sumBy({sid,lbl}, {burden})
                   )}
```

Results of the correlation filter pushed upstream

The results displayed in the table below indicate the test accuracies from using the correlation as the feature-selection based filter. We note here that, as mentioned earlier in the section, the total number of features outputted from the program was 36 000. The first column hence contains all the features, and this was done by not defining a threshold to select features.

Table 10: **Severity of prostate - Multi-omics correlation feature-selection based filter**

Method	**36000**	**20000**	**15000**
chi-square	68.6%	76.2%	59.9%
ANOVA	80.9%	80.8%	86%
RFE	-	**99.3%**	95.3%
MI	63.0%	75.1 %	71.2 %
MultiSURF	59.4%	60.9 %	53.8%

The reason why we outputted results only for 20 000 and 15 000 features, was because of how the correlation filter performed. Around 20 000 features had $|r| > 0.55$, 15 000 had $|r| > 0.90$ and 14 000 had $|r| > 0.99$. We therefore could not have used any other threshold to select a different number of features. Discussion on the performance of the correlation filter follows in Section 6.

Comparing the results above with the best performing models from the previous two experiments, RFE_10000 with test accuracy 94.0% using single-omics impact score and MI_1000 with test accuracy 87.6% using single-omics gene expression, we can see that RFE_20000 outperforms both of them with an accuracy of 99.3%. chi-square and ANOVA performed similarly, while MI's performance decreased by 10% compared to the experiment in Section 5.3.1 and around 27% compared to the experiment in Section 5.3.2. MultiSURF's performance also decreased compared to the experiment in Section 5.3.2, from 77.0% to 60.9%.

Due to the very high accuracy of RFE_20000, we decided to investigate its performance further by using different train and test splits, ensuring the very good performance was not due to randomness. We thus ran RFE_20000 4 more times using 4 different train and test splits. We obtained an average accuracy of 96.2% and using the 99.3% accuracy we obtained a final accuracy of **97.8%**. We note here that comparing with the best baseline model where no filter is pushed, ANOVA with an accuracy of 80.9%, we see an improvement of **16.9%** when we use the correlation filter.

Results of the chi-square filter pushed upstream

We then ran the same experiment using the chi-square feature-selection based filter. A benefit using this filter was that we could section the number of features much more precisely compared to the correlation filter. 20 000 features had $\tilde{\chi}^2 > 0.04$, 10 000 features had $\tilde{\chi}^2 > 0.15$, 5000 had $\tilde{\chi}^2 > 0.40$, 2000 had $\tilde{\chi}^2 > 1.10$ and 1000 had $\tilde{\chi}^2 > 2.09$.

Table 11: **Severity of prostate - Multi-omics chi-square feature-selection based filter**

Method	**20000**	**10000**	**5000**	**2000**	**1000**
chi-square	69.8%	68.8%	68.4%	67.5%	69.2%
ANOVA	80.1%	84.6%	85.4%	88.7%	84.1%
RFE	**93.5%**	92.8%	85.4%	80.0%	73.7%
MI	64.3%	61.1%	58.1%	67.2%	46.4%
MultiSURF	53.8%	54.0%	58.0%	56.8%	63.3%

We continue the experiments with an investigation into feature selection methods and filter performance on a multi-classification task. Prior to investigating multi-class, we also looked at identifying prostate tissue of origin from a pan-cancer dataset, as a more simple version of the multi-class experiment. The experiment can be found in the appendix, Section C. The experiment and the results were not thoroughly investigated due to time constraints since we decided to focus on the more challenging multi-class experiment.

5.4 Multi-class Classification: Tumor Site Prediction

In this section, we investigate pushing feature-selection based and partial filters for multi-class prediction. For this experiment, we used the TCGA data set with the MuTect annotations, discussed in Section 2.2. The data set contains information about the somatic mutations of 18 different types of cancer. For this task, we decided to select the 9 cancer types with the most samples, namely breast, endometrial, kidney, ovary, central nervous system, stomach, lung, colon, and head and neck. The neural network used was the one discussed in the introduction of the Experiments, Section 5. We present below in Figure 5.5 the data distribution of the various types of cancer (note that Cns denotes Central nervous system):

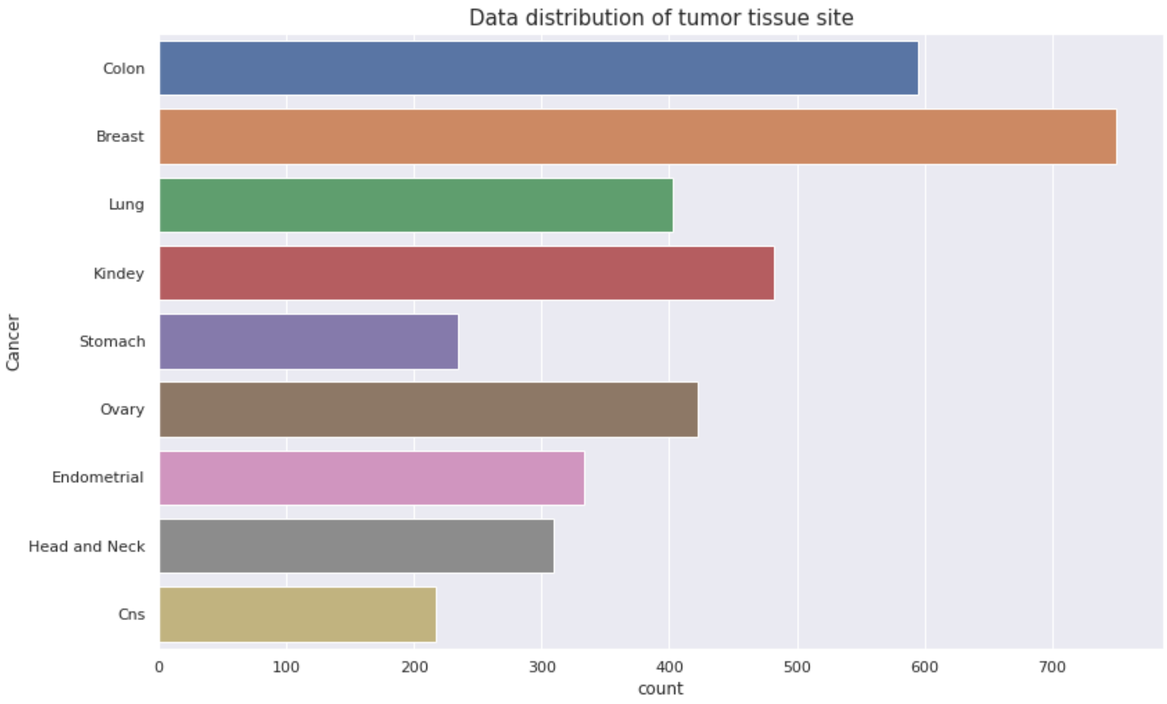


Figure 5.5: Distribution of the various cancer types

We can see that the most abundant cancer type is Breast cancer with more than x3 samples than the least abundant ones, Stomach, and Cns. Hence, we expected that the data skewness was going to have an impact on the results in the favor of Breast type cancer.

For this experiment, we used the `occurrences` data source and the single-omics mutation burden approach to calculate the GMB owing to the fact that not all 9 cancer types contained `gene_expression` data. For both experiments in this section, the `genemap`, `occurrences` and `clinical` data source were used and the size of the files were 1.2GB, 6.74 GB and 11 GB respectively.

```
GMB <=
   for g in genemap union
     {(gene:= g.g_gene_name, burdens :=
        (for o in occurrences union
           for s in clinical union
              if (o.donorId = s.sample) then
                 for t in o.transcript_consequences union
                   if (g.g_gene_id = t.gene_id) then
                      {(sid := o.donorId,
                      lbl := if (s.tumor_tissue_site = "Breast") then 1
                             else if (s.tumor_tissue_site = "Lung")
                                then 2
                             else if (s.tumor_tissue_site = "Kidney")
                                then 3
                             else if (s.tumor_tissue_site = "Stomach")
                                then 4
                             else if (s.tumor_tissue_site = "Ovary")
                                then 5
                             else if (s.tumor_tissue_site =
                                "Endometrial") then 6
                             else if (s.tumor_tissue_site = "Head and
                                Neck") then 7
                             else if (s.tumor_tissue_site = "Central
                                nervous system") then 8
                             else if (s.tumor_tissue_site = "Colon")
                                then 0,
                             else -1,
                      burden := if (t.impact = "HIGH") then 0.80
                                    else if (t.impact =
                                       "MODERATE") then 0.50
                                    else if (t.impact = "LOW")
                                       then 0.30
                                    else 0.01)}).sumBy({sid,
                                       lbl}, {burden}))}
```

The output of the program was around 24 000 features. We now look at the performance of a standard multi-class approach (Section 5.4.1) and a one-vs-rest approach (Section 5.4.2).

5.4.1 Multi-class

The aims of this experiment included the investigation of the overhead the pushed filter added to the total runtime of the program for the multi-class classification task using the simple multi-label approach. We also aimed to investigate the test accuracy of predicting the origin of cancer from 9 possible cancer types, using a single-omics approach (`occurrences`), comparing our results against an initial model that did not use a feature-selection based filter. We present below in Figure 5.6 the runtimes of this experiment.

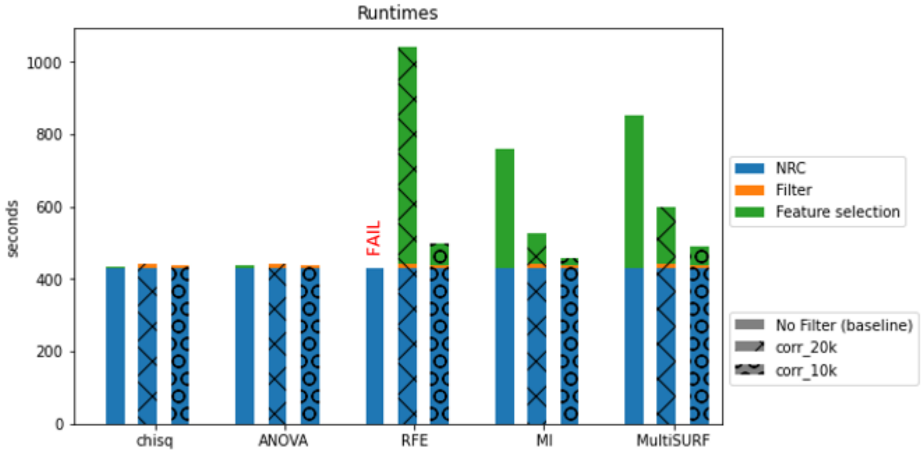


Figure 5.6: Stacked bar plot showing the runtimes of different feature selection methods of the multi-class experiment from the moment the program is executed until extracting the features.

We note that this time there are no runtimes for the chi-square filter since it was not used for this experiment. The longest runtimes are associated with RFE with quite a lot of difference. We also note that because for this experiment the calculations of the filters are more complicated by nature (9 labels present) the runtimes of Feature Selections are large compared to the ones from Figures 5.3 and 5.4. We can see that the runtimes of the correlation filter for

RFE, MI, and MultiSURF are very small compared to the total runtime while for chi-square and ANOVA, it is slightly larger.

The first column of the table below displays the results with the correlation feature-selection based filter pushed but without defining a threshold to drop any features. We also note here that we could not have used RFE using all features because of the long runtime - even though the number of features outputted compared to previous experiments was not that big, in contrast with the previous experiments the dataset was more complicated due to its multi-label structure. For this task we pushed the `corr+` filter (recall Section 4.2). This meant that initially we got rid of any features having burden values ≤ 0.01, and then used the correlation filter. After surviving the initial burden filter, 20 000 features had burden score > 0.03 had $|r| > 0.05$, 15 000 had $|r| > 0.15$ and 2000 had $|r| > 0.75$. We present the results in the table below:

Table 12: **Multi-class classification - correlation+ partial filter**

Method	**27000**	**20000**	**15000**	**2000**
chi-square	45.2%	40.2%	36.9%	32.8
ANOVA	**49.2%**	45.5%	43.7%	34.7%
RFE	-	40.9%	42.9%	35.9
MI	40.2%	37.2%	37.5 %	33.5

The best performing model was ANOVA_27000 with an accuracy of **49.2%**. In this case, the filter does not help with the test accuracy of any of the models, as with very few exceptions the accuracies decline when using fewer features. A potential explanation for the relatively low accuracy first was the task at hand. Multi-class classification with 9 different cancer types is a challenging task. Furthermore, the total number of samples was 3499, which were split for training, validation, and testing. For such a challenging task the models perhaps required more samples to train and by removing features when using the filters, we lose valuable predictors. The result however is an improvement of the results presented in the original paper, [37], which was 42.32%.

5.4.2 One-vs-rest

The aims of this experiment included the investigation of test and training accuracies of predicting the origin of cancer from 9 possible cancer types as described before. The one-vs-rest classification method concerned the same multi-class classification task. The experiment was performed with the same dataset and program as the one in the previous Section, 5.4.1. For this experiment, however, the way we made predictions was different. First, we trained 9 *binary* models, one for each cancer type independently. The only difference in the neural network we used for this task compared to the multi-class problem was that the loss function was binary cross-entropy and we used a sigmoid output activation function. Each one of the 9 models outputted the probability of a patient developing that particular cancer. We then merged the models to perform our predictions. The models were merged in terms of the probabilities they outputted and the prediction was made using the classifier with the highest probability for that particular patient. If for example, the colon model outputted a probability of 0.85 for a particular patient having colon cancer, while the other 8 models outputted a probability of < 0.85 for their respective cancer types, the prediction we made was that the patient had colon cancer. We now discuss the different ways we calculated the accuracy of the models. The differences concern the way we trained the models.

Training accuracy experiment:

The first task was very similar to the one presented in the original paper, [37]. More specifically, we used all 3499 samples with a 70% and 30% training and validation split for training. For each of the 9 classifiers, we used the feature selection methods below to output a feature set independent of the other classifiers. For example, when we write results for the ANOVA feature selection method, each time we trained a binary classifier, we ran ANOVA for each classifier independently. Hence, each binary classifier potentially had a different feature set as its input (again topk_features = 200). Also, what we report below, is the *training accuracy.* After we trained each individual classifier on the training dataset, we merged the classifiers as discussed and checked the performance on all 3499 samples. The classifiers were trained for 30 epochs, and the training accuracies reported is an average over 5 runs. Initially, we used all features (24 000) without any filter pushed. The average for chi-square was 74.6% and for ANOVA 75.7%.

<u>Source and value filters</u>:

In order to investigate RFE, we decided to initially push the source filter, discussed in Section 4.2. The threshold we used for outputting 12 000 in column 1 was 0.01, and so we filtered out any genes with impact scores < 0.01, i.e. removed the "LOW" impact genes. For the next two columns below, the filter used was the value filter (see Section 4.2). We defined a threshold of 3.6 where 6500 features survived and 7.6 where 3500 features survived. The results shown below were obtained again from 30 epochs and averaging over 5 runs:

Table 13: **Multi-class classification - one-vs-rest - value filter**

Method	**12000**	**6500**	**3500**
chi-square	73.7%	65.1%	55.1%
ANOVA	77.0%	66.5%	56.2%
RFE	**80.3%**	71.7%	62.1%

We see that the filter slightly improved the performance of ANOVA and decreased the performance of chi-square. More importantly, RFE was now able to run outputting the best accuracy, **80.3%** (2% improvement on the result reported in the original paper, [37]). In the Figure below, 5.7, we display the training accuracy for all the binary classification models on the same graph. We can see that for training, with relatively few epochs the classifiers are able to perform well on their respective binary classification task.

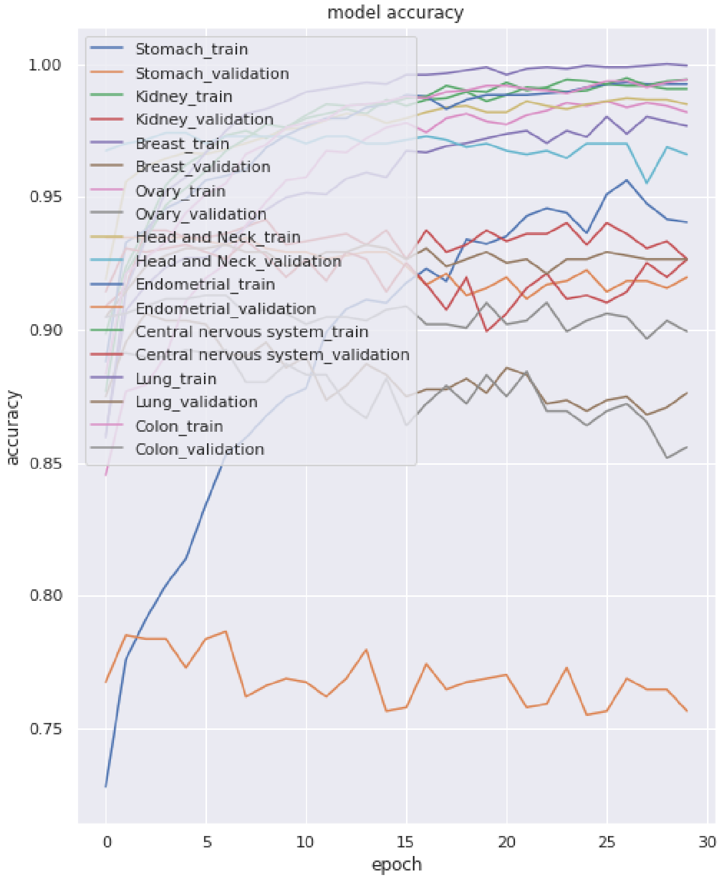
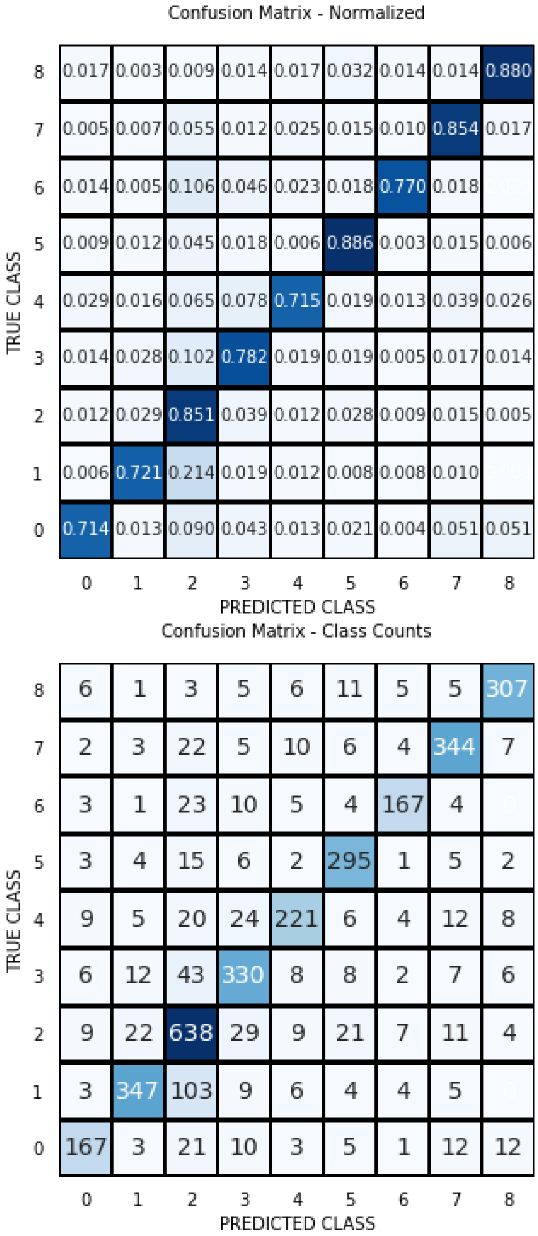


Figure 5.7: Multi-class classification - one-vs-rest approach. The accuracies displayed are the training accuracies of all binary classifiers

We also present below in Figure 5.8, the confusion matrix using RFE_12000. The final accuracy is the sum of the entries in the diagonal over the total number of entries in the Class Counts graph. In the diagonal of the Normalized graph, the accuracies for each cancer type are displayed. False Positives and False Negatives rates are displayed in the off-diagonal entries.



Confusion Matrix - Normalized

TRUE CLASS	0	1	2	3	4	5	6	7	8
8	0.017	0.003	0.009	0.014	0.017	0.032	0.014	0.014	0.880
7	0.005	0.007	0.055	0.012	0.025	0.015	0.010	0.854	0.017
6	0.014	0.005	0.106	0.046	0.023	0.018	0.770	0.018	0.000
5	0.009	0.012	0.045	0.018	0.006	0.886	0.003	0.015	0.006
4	0.029	0.016	0.065	0.078	0.715	0.019	0.013	0.039	0.026
3	0.014	0.028	0.102	0.782	0.019	0.019	0.005	0.017	0.014
2	0.012	0.029	0.851	0.039	0.012	0.028	0.009	0.015	0.005
1	0.006	0.721	0.214	0.019	0.012	0.008	0.008	0.010	0.000
0	0.714	0.013	0.090	0.043	0.013	0.021	0.004	0.051	0.051

PREDICTED CLASS

Confusion Matrix - Class Counts

TRUE CLASS	0	1	2	3	4	5	6	7	8
8	6	1	3	5	6	11	5	5	307
7	2	3	22	5	10	6	4	344	7
6	3	1	23	10	5	4	167	4	0
5	3	4	15	6	2	295	1	5	2
4	9	5	20	24	221	6	4	12	8
3	6	12	43	330	8	8	2	7	6
2	9	22	638	29	9	21	7	11	4
1	3	347	103	9	6	4	4	5	0
0	167	3	21	10	3	5	1	12	12

PREDICTED CLASS

Figure 5.8: Confusion matrix of the RFE model. The cancer types are encoded as [Colon = 0, Breast = 1, Lung = 2, Kidney = 3, Stomach = 4, Ovary = 5, Endometrial = 6, Head and Neck = 7, Central nervous system = 8]

In contrast with the results we obtained in the binary classification task, from the results of table 13, we can see that the performance of the models greatly decreases by using this filter. Biologically, this makes sense. As we filter by increasing the burden threshold, we are left with genes with high burdens and that implies that the higher the burden the more mutations and hence the more aggressive cancer they represent. A high burden hence implies a more individually specific cancer type. Therefore, for the one-vs-rest task, these genes are not helpful as they provide good information which is cancer-specific and so is not shared across the rest cancer types. Hence, when merging the results of the binary models, we have some genes that are very confident in predicting specific cancers for the binary classifiers but that are not useful when merging together probability results for the cancer types that the genes are not confident in predicting.

Testing accuracy experiment:

The final experiment we considered was investigating the performance of the one-vs-rest classifiers for the multi-class classification task in a more standard training and testing setting. For this experiment, we trained the classifiers using 70% of the data (which was further split into training and validation sets), and tested on the other 30% of data. We used all features outputted by the program (24 000) without pushing any filters. Due to time constraints and the fact that ANOVA feature selection runtime was very short, we performed the experiment using ANOVA only (which was close to being the best performing model for the training accuracy experiment and was the best model from the multi-class classification problem in Section 5.4.1). Initially, as with the case of the training accuracy experiment, we ran ANOVA for each classifier independently, selecting topk_features = 200 for each. The average *testing* accuracy over 5 runs and 30 epochs each was **52.7%**.

We then considered different feature selection approaches for the same task. Using ANOVA as the feature selector throughout, we outputted topk_features = 20 for each cancer type and concatenated them together resulting in 180 features before training the binary classifiers. We then trained them using this common feature set of 180. The average accuracy was 52.0%.

We then performed the same experiment but with the common feature set being the same as the above, but excluding the intersecting features between the binary classifiers, resulting in a total of 123 features. The average accuracy was 41.2%. Finally, we obtained an accuracy of 41.8% when running the experiment with the feature set containing only the intersecting features (21).

The best performing feature set was the one where we trained each classifier independently, achieving an accuracy of 52.7%. The relatively low accuracy, however, implies that the task at hand needs a more complicated feature selection method or a different approach when calculating the mutational burden. Regardless, this accuracy is an improvement compared to the simpler multi-class classification results (ANOVA with 49.2%) and a 10% improvement on the results reported in the original paper, [37].

5.5 Gene Enrichment Analysis

In this section, we present our findings on the analysis of the gene output from the models in the previous experiments. We selected gene sets from the best performing models for the prostate and multi-site [10] analysis from above in order to gain more insight into what these genes say about the biological process and to gain confidence in the feature sets we are creating. We performed gene set enrichment analysis on these gene sets using WebGestalt, [53]. Gene set enrichment is a method to measure the correlation of an input gene set to existing gene sets. We used parameters from the interface of the website specific to our task. We had gene ontology as a functional database, genome as our reference list, and Over-Representation Analysis (ORA) for our methods of interest. ORA is a statistical method that determines whether genes from pre-defined sets (e.g. from the genome list from WebGestalt) are present more than it would be expected (over-represented) in the data we test. For more details, we refer the reader to [32]. We provide the categorical organization of the genes within each set with respect to biological processes, cellular components, and molecular function. We also discuss gene set enrichment results, noting any gene sets that were identified with False-Discovery Rate (FDR) ≤ 0.05. 0.05 is a generally accepted threshold in biology. FDR is the expected proportion of false positives among the rejected hypotheses. For this application, the null hypothesis is that the gene sets are independent of the biological processes we identify and the alternative hypothesis is that there is a significant correlation between the gene set and the biological processes. FDR ≤ 0.05 implies that less than 5% of all tests will result in false positives, therefore we are more than 95% confident that our results are accurate and that the gene set is correlated with the biological processes.

5.5.1 Prostate severity with multi-omics

From the experiment in Section 5.3.3, we extracted genes from the best-performing models. This was from the correlation feature-selection based filter outputting 20 000 features (middle column of table 10). We analyzed the full output, i.e. all 200 features, of RFE (which

---

[10] Multi-class classification

was the best performing model). We then analyzed the overlap of the output genes from RFE/ANOVA (best vs second-best model). Finally, we considered the genes outputted by RFE but excluding the genes overlapping with ANOVA. Recall that each model outputted 200 genes. We found 52 overlapping genes between ANOVA and RFE and therefore 148 genes identified by RFE only.

The gene enrichment analysis for the full output of RFE (200 genes) did not generate any high-confidence gene set enrichment scores (all had $FDR > 0.05$). We did identify two gene sets that were identified with $FDR < 0.05$ in the overlapping set of RFE and ANOVA, both of them specific to metabolic processes. The full set from the overlapping feature set is presented in Figure 5.9. We discuss the significance of this further in the Discussion, Section 6.

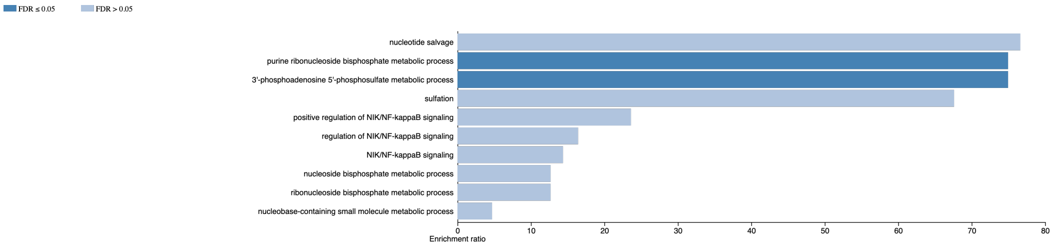


Figure 5.9: Gene set enrichment analysis for overlapping RFE/ANOVA feature sets. High-confidence associations are in dark blue. Generated by WebGestalt.

The gene ontology classification summary for the overlapping gene set is provided in Figure 5.10. We provide only the overlap since the others did not report high-confidence enrichment results. The other classification summaries for this and that other method can be found in the appendix, D. For the overlapping set, we can see that metabolism was identified as the most represented biological process - consistent with the high-confidence enrichment results. Cellular components (membrane, nucleus) and molecular functions (protein binding) are consistent with cancer-specific processes. These results suggest that the multi-omics approaches are able to identify prostate-cancer-specific processes like metabolism as well as features that are specific to cancer in general.

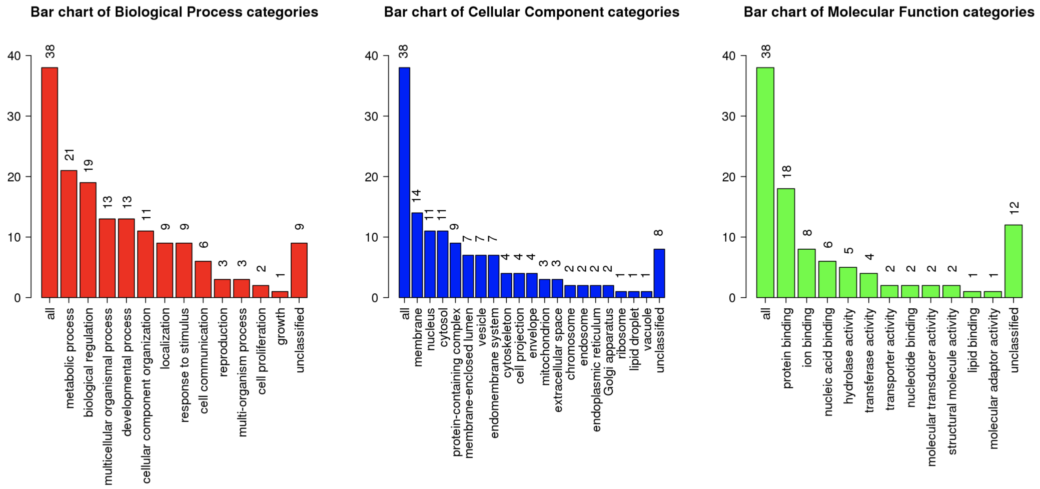


Figure 5.10: Represented biological processes, cellular components, and molecular function categories for the overlapping gene set of ANOVA/RFE. Generated by WebGestalt.

5.5.2 One-vs-rest Tumor Site Prediction

For the best performing model from the tumor site prediction problem, we took the gene set from the best performing model (ANOVA, partial filter > 0.01) and (ANOVA with no filter). While the training accuracy was highest with the partial filter, their testing accuracies were the same within 0.7%. We looked at the overlap between these two feature sets to determine if the low testing accuracy was related to the feature selection method making random decisions from an ill-informed feature space. We found 114 overlapping genes, suggesting that the features selected aren't just due to random chance. The gene enrichment analysis for the pan-cancer case study surprisingly, based on what we saw from model performance, had higher confidence results than the prostate analysis. Ten gene sets were identified with FDR ≤ 0.05, which are noted in the volcano plot in Figure 5.11. The volcano plot shows the -log of FDR vs the enrichment ratio, highlighting the degree by which the significant categories stand out (upper corners).

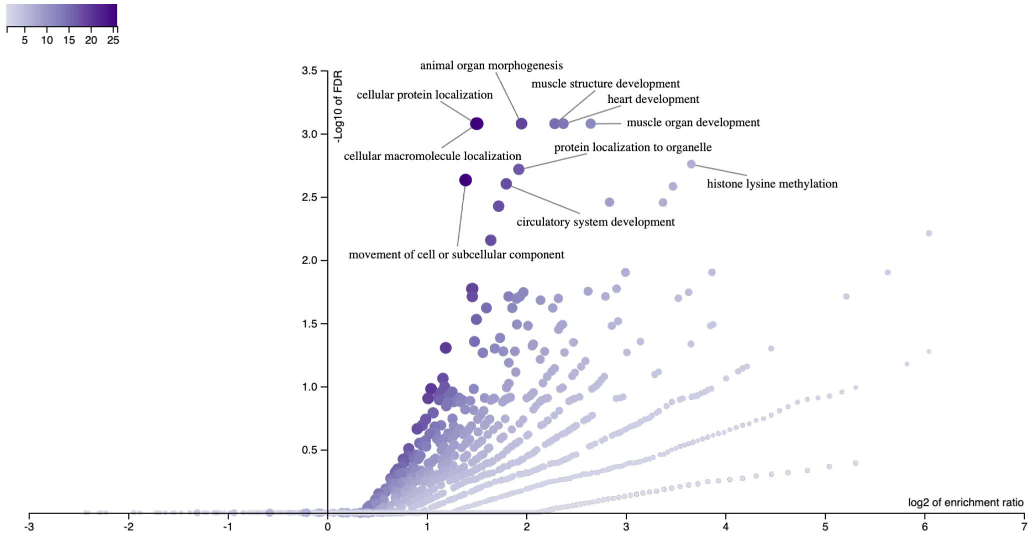


Figure 5.11: Volcano plot for the 114 overlapping ANOVA genes used in the one-vs-rest experiment. High-confidence associations are labeled by definition. Generated by WebGestalt.

The gene ontology classification summary is provided in 5.12. 89 out of the 114 genes were successfully mapped to publically available gene sets, with relatively low amounts of classifications. There is a high representation of genes in many biological processes that one would expect to be associated with cancer, such as biological regulation and developmental processes. This is true for cellular component categories as well, with nucleus and membrane being the most represented group. Finally, the most represented molecular function is protein-binding which is also in line with cancer since many systems will have disruptions in regulation due to binding issues. These results highlight how the ANOVA method was able to pick up processes that are shared across cancer types.

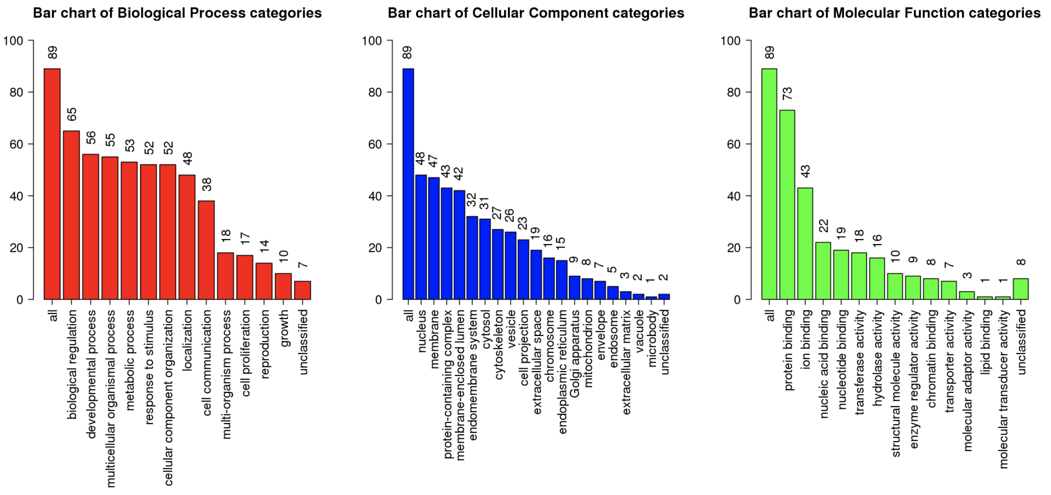


Figure 5.12: Represented biological processes, cellular components, and molecular function categories for the 114 overlapping ANOVA genes used in one-vs-rest.

Section summary:

From the runtimes experiments we performed, we first showed that pushing feature-selection based filters from UDFs into the NRC program speeds up feature extraction for the majority of feature selection methods we considered. Second, they enable RFE to run by reducing the large feature space. Moreover, the filters add minimal overhead to the total runtime of the NRC program. The best performing model for the binary classification task of predicting the severity of prostate cancer was 97.8% using RFE and a multi-omics approach with the correlation filter pushed. Comparing for the multi-class classification task, RFE was the best performing model for the training accuracy experiment with an accuracy of 80.3%, while for the testing accuracy experiment ANOVA outputted an accuracy of 52.7%. We then analyzed features outputted by the models by performing gene enrichment analysis. For the binary classification task (severity of prostate cancer), we discovered genes with correlation to specific metabolic processes from the overlapping set of features from the two best performing models, RFE and ANOVA. For the multi-class classification task, from the one-vs-rest approach, we successfully mapped gene sets to publically available genes sets and associated them with cancer-specific biological processes such as cell regulation. Finally, we showed that for all the binary classification methods we considered, we increased the performance (test accuracy) of the best performing baseline model with the filters we used, with the highest increase in performance being 16.9%.

6 Discussion and Future Work

We now discuss some key points, findings, limitations, and challenges we faced throughout the thesis. We will focus the discussion on the experimental results from Section 5.

6.1 Discussion

From the runtime bar plots in Figures 5.3, 5.4 and 5.6 we can see that the runtimes for the feature-selection based filters, chi-square and correlation, are very small compared to the total runtime. The total runtime includes the time taken for the NRC program to run until the time the features are selected by the feature selection methods, NRC + Filter + Feature selection. The longer runtimes occur when extracting the features, mainly using RFE, MI, and MultiSURF. We can therefore see that the feature-selection based filters add minimal overhead for the training pipeline runtime and that for many experiments (e.g. the multi-omics experiment in Section 5.3.3) they also improve the performance of the models. For all the experiments, they also enable the use of RFE which ends up being the best performing model on many occasions.

In experiments involving point mutations, for example in Section 5.3.1, we noticed that the correlation filter was not able to select features in a uniform way, outputting skewed results with many features having correlations very close to +1 or -1 (within 0.01). Recall the equation of correlation:

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

To provide insight as to why this was the case, we note that since there are many features with burden values very close to 0 (features with no impact values getting a value of 0.01 from the program) the moduli of the numerator and the denominator of the calculation above are very similar. Since the mean of each gene, $\bar{x}$, is very small for many genes having small burden scores, this implies that $(x_i - \bar{x})$ is a very small number as well, and hence the outcomes of the two calculations: $\sum_{i=1}^{n}(x_i - \bar{x})$ and $\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}$ have very similar moduli. Therefore, the calculation boils down to $\frac{\sum_{i=1}^{n}(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$. Because the values of the labels are in many instances 0, that means that in many cases the modulus of the fraction is also very close to 1. Therefore, overall outputting a correlation value very close to 1 or -1. Even though the correlation between point mutations and their burden (what we just described) isn't as informative, mixing with gene expression (i.e. Fpkm) values is beneficial. As we saw in Table 8, we can split features more precisely when using gene expression values compared to mutation impact values (occurrences). Impact burden values are additive for a gene but gene expression has a one-to-one correspondence to a gene (one Fpkm value per gene). So, correlation filter methods are not as strong when using impact values and therefore when integrating more datasets, which is why we can't differentiate datasets very well with the correlation filter for the multi-omics task. Despite this challenge, as we show in experiment 5.3.2, gene expression does not perform well on its own compared to impact values, 5.3.1, so impact values on their own appear to be more useful for the binary classification task on prostate cancer severity. However, the fact that the best performing model is observed when using a multi-omics approach, it implies that the correlation filter we push was very vital on the difference in the performance of the model.

From the results of the experiments, we note that the best performing model for the binary classification task of predicting the severity of prostate cancer is RFE with 20 000 features with an accuracy of 97.8%. The fact that this model was able to run because of the correlation feature-selection based filter we used highlights the importance of the filter. Furthermore, the fact that the GMB calculation for this experiment was done with a multi-omics approach, by combining `occurrences` and `gene_expression`, shows the significance of integrating different data sources and motivates further exploration on using such models to identify complex patterns.

The best performing model for the training accuracy experiment of the multi-class classification problem is from the one-vs-rest approach. Namely, RFE with 12 000 features with an accuracy of 80.3% where the value filter was pushed. The value filter was used because the correlation and chi-square filter took too long to execute owing to the small cluster size of the computer node. For the testing accuracy experiment of the multi-class classification task (again one-vs-rest approach), ANOVA is the best performing model using all the features, with an accuracy of 52.7%. The multi-class classification task we considered however is a challenging task by its nature. Feature selection methods from Python packages, `Python.sklearn` [46], work by binarizing the target variable if more than 2 labels are present. The fact that this task involves 9 labels results in even more estimations for calculating feature importance scores. In addition, the sample size we used was fairly small, with 3499 samples. Therefore, methods that are capable of taking into account the complexities of the biological system, the relationship between and across cancers, and gene-gene relationships will perform better to such challenging tasks. This will mean determining more advanced filters that can be pushed for more complicated feature selection methods. The gene enrichment analysis below further supports this claim.

For the prostate severity problem (binary classification), though we do not see many high-confidence gene sets in the enrichment analysis, we do see categories that are different from what we have seen in the pan-cancer (multi-class classification) analysis. This implies that we are definitely identifying genes that are important for prostate cancer, which is backed up by model performance. The interesting thing is that the overlap of the top-performing models (RFE and ANOVA) was the only gene set that provided a high-confidence association to a known gene set, which was related to metabolic processes. This is supported by recent research which shows that there are specific metabolic processes driving tumor development in prostate cancer, where metabolic drugs have been suggested as a treatment option for prostate cancer [34]. The other gene sets that do not have high confidence in enrichment analysis, but high performance in prediction, suggest that the important genes are associated with many biological systems but do not come across in a gene set enrichment. The features should be analyzed with respect to all the systems involved rather than looking for multiple overlapping genes within the same biological system. This also suggests that multi-omics approaches that will consider more information from the system as a whole will be beneficial for predicting prostate severity, and the high accuracy of our multi-omics model further supports this.

For the multi-class classification task, 5.4.2, as noted in the experiments, ANOVA is able to pick up many biological processes, cellular components, and molecular functions that are common in cancer. One interesting process that stood out was the high categorization of nucleus and membrane in the cellular component category. The nuclei of cancer cells are often much larger and darker than non-cancerous cells and are used as a cancer diagnosis and progression marker in tumor tissue slides. Further, it has recently been shown that the nuclear envelope plays an important role in cancer given its role in cell division and gene regulation [30]. If we consider the categorical results in combination with the accuracies reported in one-vs-rest, the results suggest that the feature selection methods are identifying features that are important across all cancers. The best performing model for the one-vs-rest task, ANOVA, works by reducing the variance between the features and the target by selecting features that discriminate the most between the two. This is done for each binary classifier. These features are the ones that are more "easy" to pick up, and so are shared across different cancer types, as they are common to all.

The patterns we see in the selection of features common across all cancer type in the multi-class problem also explains why the pan-cancer feature selection, which leads to poor model performance, has higher-confidence features in gene enrichment than for the prostate cancer analysis, which has high model performance. The feature selection for pan-cancer understands the problem as if it is faced with an easier task - to identify features that are associated with cancer tissues, and ends up doing well. This, however, is masking within cancer-specific features that we need to tackle for the multi-class classification task. The prostate cancer analysis, however, has a more specific question - find a specific set of genes that tells us how severe the cancer is, and as it turns out the feature selection methods can do this and produce high performing models, but the relationships between the genes in these feature sets are less clear. This is likely because their relationship is not one overarching system (like cancer in the multi-class problem) at play. The significant features that stand out in the prostate analysis are forming more complex relationships that span many processes, making it impossible for gene set enrichment methods to identify their relationships with high confidence.

We see that the feature-selection based filters provide minimal overhead for the training pipeline and overall improve the performance of our models. Overall, we see that the binary classification task benefits from the pushed filters but more exploration is needed for the multi-class prediction. The multi-class prediction task is a much harder task, and through the gene enrichment analysis, we showed that even though our model was able to pick up cancer-specific biological processes, the performance of the model in terms of test accuracy is still relatively low.

6.2 Limitations and Future Work

We now discuss some limitations of our approach and suggest improvements. We first note that the results we present were tested on a single dataset (TCGA), and hence further experiments need to be run using a variety of datasets to increase the confidence of our results. Moreover, the experiments need to be executed on a larger cluster in order to assess scalability of UDF-based filters since they benefit from the computer node distribution as discussed in Section 4.1.3. This will also benefit situations where we were unable to run programs on larger datasets and needed to use source and value filters to reduce the initial feature space. Also, we recall in Figure 3.2, that Feature extraction happens in external libraries. The filters we introduce, feature-selection based filters and source and value filters (recall Section 4.1), are applied to the materialized results of the NRC program representing feature matrix construction. This allows us to implement the filters, previously only done locally, to a distributed setting. Both the application of the filter and the distributed implementation of the feature-selection based filter will speed up the total execution of the NRC program. We suggest further investigation to push more of the feature-selection based filter calculation into the NRC program representing the feature matrix construction to speed up matrix construction.

More specific to the feature-selection use case, the binarization of target variables is not yet implemented in the filters we introduced. For the chi-square feature-selection based filter for the multi-class classification, this meant that we had to manually code the binarization. We therefore suggest automating the binarization as that will greatly increase the use cases of chi-square and remove the burden from programmers to manually binarize the target variable. Finally, we note that the filters we implement and the experiments we run are only done for the shredded pipeline. This is because past work has shown that shredding will always outperform the standard compilation route [37]. Investigating this for the standard pipeline as well might identify interesting feature-based optimizations that can be applied regardless of the underlying representation.

In addition to the suggestions above, we now discuss areas of future work that we think will be beneficial for UDFs and biomedical classification tasks. First, we discuss extensions for the UDFs, such as extending UDFs to support Integrated Development Environment, IDEs. These might involve processes such as exception handling, memory management, and Web UI (User Interface) to allow the user to monitor the pipeline of the UDF execution during the time of execution. This will allow users to have a better understanding of the runtimes of processes executed by the UDFs and manage them more efficiently. Recent research in the area of UDFs has already implemented some of these, such as the Tuplex analytics framework [44].

We now suggest further investigations regarding the applications of TraNCE and biomedical classification tasks:

- We suggest investigating other prediction tasks (even for different diseases), both binary and multi-class, to check whether the performance patterns we saw in our experiments are common with the new tasks. It will be interesting to investigate the performance of the feature-selection based filters for other tasks and see whether we can draw similar conclusions for tasks not relating to cancer

- Due to the promising results from the multi-omics approach for binary classification tasks relating to cancer, we suggest integrating more or different data sources to tackle the more challenging multi-class classification task.

- Following our discussion on the challenges the feature selection methods we considered face for the multi-class classification task, we suggest different approaches that use more recent and sophisticated methods for extracting features from large feature spaces. These might include graph embedding approaches, such as Graph Neural Networks (GNNs), that have recently shown very promising results in biomedical analyses [45, 33].

7 Conclusions

In this thesis, we have successfully extended the query compilation framework TraNCE, by extending NRC with statistical operations through the integration of UDFs to support them. Furthermore, we have successfully implemented a hint optimization which enables users defining the UDFs to define thresholds and push filters in order to optimize the execution of the UDFs and reduce the large feature space that often exists in biomedical tasks. We have also managed to improve testing accuracy benchmarks for the binary classification task of predicting the severity of prostate cancer by using a multi-omics approach and the training accuracy benchmark for the multi-class classification task. Finally, we have analyzed gene sets and their relevance in biological processes and concluded interesting findings.

References

[1] Steven D. Carson. "BASIC program for non-parametric fitting of user-defined functions to experimental data with plotting of results". In: *Computer Methods and Programs in Biomedicine* 29.4 (1989), pp. 229–234. ISSN: 0169-2607. DOI: `https://doi.org/10.1016/0169-2607(89)90107-7`. URL: `https://www.sciencedirect.com/science/article/pii/0169260789901077`.

[2] Limsoon Wong. "Querying nested collections". In: (1994). URL: `https://repository.upenn.edu/dissertations/AAI9503855`.

[3] Jan Van den Bussche. *Simulation of the Nested Relational Algebra By the Flat Relational Algebra, With an Application to the Complexity of Evaluating Powerset Algebra Expressions*. 1999.

[4] Leonidas Fegaras and David Maier. "Optimizing Object Queries Using an Effective Calculus". In: *ACM Trans. Database Syst.* 25.4 (Dec. 2000), pp. 457–516. ISSN: 0362-5915. DOI: `10.1145/377674.377676`. URL: `https://doi.org/10.1145/377674.377676`.

[5] LIANGYOU CHEN and HASAN M. JAMIL. "ON USING REMOTE USER DEFINED FUNCTIONS AS WRAPPERS FOR BIOLOGICAL DATABASE INTEROPERABILITY". In: *International Journal of Cooperative Information Systems* 12.02 (2003), pp. 161–195. DOI: `10.1142/S021884300300070X`. eprint: `https://doi.org/10.1142/S021884300300070X`. URL: `https://doi.org/10.1142/S021884300300070X`.

[6] Jianping Hua et al. "Optimal number of features as a function of sample size for various classification rules". In: *Bioinformatics* 21.8 (Nov. 2004), pp. 1509–1515. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/bti171`. eprint: `https:`

//academic.oup.com/bioinformatics/article-pdf/21/8/1509/691983/bti171.pdf. URL: https://doi.org/10.1093/bioinformatics/bti171.

[7] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[8] Ivan A. Adzhubei et al. "A method and server for predicting damaging missense mutations". eng. In: *Nature methods* 7.4 (Apr. 2010). nmeth0410-248[PII], pp. 248–249. ISSN: 1548-7105. DOI: 10.1038/nmeth0410-248. URL: https://doi.org/10.1038/nmeth0410-248.

[9] Alexander Alexandrov et al. "Massively Parallel Data Analysis with PACTs on Nephele". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 1625–1628. ISSN: 2150-8097. DOI: 10.14778/1920841.1921056. URL: https://doi.org/10.14778/1920841.1921056.

[10] Idan Menashe et al. "Pathway analysis of breast cancer genome-wide association study highlights three pathways and one canonical signaling cascade". eng. In: *Cancer research* 70.11 (June 2010). 0008-5472.CAN-09-4502[PII], pp. 4453–4459. ISSN: 1538-7445. DOI: 10.1158/0008-5472.CAN-09-4502. URL: https://doi.org/10.1158/0008-5472.CAN-09-4502.

[11] Michael F. Ochs. "Genomics Data Analysis Pipelines". In: *Biomedical Informatics for Cancer Research*. Ed. by Michael F. Ochs, John T. Casagrande, and Ramana V. Davuluri. Boston, MA: Springer US, 2010, pp. 117–137. ISBN: 978-1-4419-5714-6. DOI: 10.1007/978-1-4419-5714-6_6. URL: https://doi.org/10.1007/978-1-4419-5714-6_6.

[12] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: USA: USENIX Association, 2010.

[13] Peilin Jia, Yang Liu, and Zhongming Zhao. “Integrative pathway analysis of genome-wide association studies and gene expression data in prostate cancer”. eng. In: *BMC systems biology* 6 Suppl 3.Suppl 3 (2012). 1752-0509-6-S3-S13[PII], S13–S13. ISSN: 1752-0509. DOI: `10.1186/1752-0509-6-S3-S13`. URL: `https://doi.org/10.1186/1752-0509-6-S3-S13`.

[14] Vijay K. Ramanan et al. “Pathway analysis of genomic data: concepts, methods, and prospects for future development”. In: *Trends in Genetics* 28.7 (2012), pp. 323–332. ISSN: 0168-9525. DOI: `https://doi.org/10.1016/j.tig.2012.03.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0168952512000364`.

[15] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.

[16] Jiaxing Zhang et al. “Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions”. In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 295–308. ISBN: 978-931971-92-8. URL: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zhang`.

[17] Claudia Beleites et al. “Sample size planning for classification models”. In: *Analytica Chimica Acta* 760 (2013), pp. 25–33. ISSN: 0003-2670. DOI: `https://doi.`

org/10.1016/j.aca.2012.11.007. URL: https://www.sciencedirect.com/science/article/pii/S0003267012016479.

[18] Kristian Cibulskis et al. “Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples”. In: *Nature Biotechnology* 31.3 (Mar. 2013), pp. 213–219. ISSN: 1546-1696. DOI: 10.1038/nbt.2514. URL: https://doi.org/10.1038/nbt.2514.

[19] “The cancer genome atlas pan-cancer analysis project”. English (US). In: *Nature Genetics* 45.10 (Oct. 2013), pp. 1113–1120. ISSN: 1061-4036. DOI: 10.1038/ng.2764.

[20] Chen Meng et al. “A multivariate approach to the integration of multi-omics datasets”. In: *BMC Bioinformatics* 15.1 (May 2014), p. 162. ISSN: 1471-2105. DOI: 10.1186/1471-2105-15-162. URL: https://doi.org/10.1186/1471-2105-15-162.

[21] Adam Auton et al. “A global reference for human genetic variation”. English (US). In: *Nature* 526.7571 (Sept. 2015), pp. 68–74. ISSN: 0028-0836. DOI: 10.1038/nature15393.

[22] William McLaren et al. “The Ensembl Variant Effect Predictor”. In: *Genome Biology* 17.1 (June 2016), p. 122. ISSN: 1474-760X. DOI: 10.1186/s13059-016-0974-4. URL: https://doi.org/10.1186/s13059-016-0974-4.

[23] John H. Phan et al. “Integration of Multi-Modal Biomedical Data to Predict Cancer Grade and Patient Survival”. eng. In: *... IEEE-EMBS International Conference on Biomedical and Health Informatics. IEEE-EMBS International Conference on Biomedical and Health Informatics* 2016 (Feb. 2016). PMC4969000[pmcid], pp. 577–580. ISSN: 2641-3590. DOI: 10.1109/BHI.2016.7455963. URL: https://doi.org/10.1109/BHI.2016.7455963.

[24] *Apache Zeppelin*. June 2017. URL: `https://zeppelin.apache.org`.

[25] Zhaoyi Chen et al. “Trends in Gene Expression Profiling for Prostate Cancer Risk Assessment: A Systematic Review”. eng. In: *Biomedicine hub* 2.2 (May 2017). bmh-0002-0001[PII], pp. 1–15. ISSN: 2296-6870. DOI: `10.1159/000472146`. URL: `https://doi.org/10.1159/000472146`.

[26] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. “Optimization of Complex Dataflows with User-Defined Functions”. In: *ACM Comput. Surv.* 50.3 (May 2017). ISSN: 0360-0300. DOI: `10.1145/3078752`. URL: `https://doi.org/10.1145/3078752`.

[27] Huijskens Thomas. *Mutual information-based feature selection*. `https://thuijskens.github.io/2017/10/07/feature-selection/`. Oct. 2017.

[28] Danilo Bzdok et al. “Prediction and inference diverge in biomedicine: Simulations and real-world data”. In: *bioRxiv* (2018). DOI: `10.1101/327437`. eprint: `https://www.biorxiv.org/content/early/2018/05/21/327437.full.pdf`. URL: `https://www.biorxiv.org/content/early/2018/05/21/327437`.

[29] Ryan J. Urbanowicz et al. “Benchmarking relief-based feature selection methods for bioinformatics data mining”. In: *Journal of Biomedical Informatics* 85 (2018), pp. 168–188. ISSN: 1532-0464. DOI: `https://doi.org/10.1016/j.jbi.2018.07.015`. URL: `https://www.sciencedirect.com/science/article/pii/S1532046418301412`.

[30] Maria Alvarado-Kristensson and Catalina Ana Rosselló. “The Biology of the Nuclear Envelope and Its Implications in Cancer Biology”. eng. In: *International journal of molecular sciences* 20.10 (May 2019). ijms20102586[PII], p. 2586. ISSN: 1422-0067. DOI: `10.3390/ijms20102586`. URL: `https://doi.org/10.3390/ijms20102586`.

[31] Laura Fancello et al. “Tumor mutational burden quantification from targeted gene panels: major advancements and challenges”. In: *Journal for ImmunoTherapy of Cancer* 7.1 (2019). DOI: `10.1186/s40425-019-0647-4`. eprint: `https://jitc.bmj.com/content/7/1/183.full.pdf`. URL: `https://jitc.bmj.com/content/7/1/183`.

[32] Yuxing Liao et al. “WebGestalt 2019: gene set analysis toolkit with revamped UIs and APIs”. In: *Nucleic Acids Research* 47.W1 (May 2019), W199–W205. ISSN: 0305-1048. DOI: `10.1093/nar/gkz401`. eprint: `https://academic.oup.com/nar/article-pdf/47/W1/W199/28880211/gkz401.pdf`. URL: `https://doi.org/10.1093/nar/gkz401`.

[33] Jaechang Lim et al. “Predicting Drug–Target Interaction Using a Novel Graph Neural Network with 3D Structure-Embedded Graph Representation”. In: *Journal of Chemical Information and Modeling* 59.9 (Sept. 2019), pp. 3981–3988. ISSN: 1549-9596. DOI: `10.1021/acs.jcim.9b00387`. URL: `https://doi.org/10.1021/acs.jcim.9b00387`.

[34] David A. Bader and Sean E. McGuire. “Tumour metabolism and its unique properties in prostate adenocarcinoma”. In: *Nature Reviews Urology* 17.4 (Apr. 2020), pp. 214–231. ISSN: 1759-4820. DOI: `10.1038/s41585-020-0288-x`. URL: `https://doi.org/10.1038/s41585-020-0288-x`.

[35] Tudor Baetu. *Causal Inference in Biomedical Research*. 2020. URL: `http://philsci-archive.pitt.edu/17674/`.

[36] Andrea Bommert et al. “Benchmark for filter methods for feature selection in high-dimensional classification data”. In: *Computational Statistics and Data Analysis* 143 (2020), p. 106839. ISSN: 0167-9473. DOI: `https://doi.org/10.1016/`

j.csda.2019.106839. URL: https://www.sciencedirect.com/science/article/pii/S016794731930194X.

[37] Jaclyn Smith et al. "Scalable Analysis of Multi-Modal Biomedical Data". In: *bioRxiv* (2020). DOI: 10.1101/2020.12.14.422781. eprint: https://www.biorxiv.org/content/early/2020/12/15/2020.12.14.422781.full.pdf. URL: https://www.biorxiv.org/content/early/2020/12/15/2020.12.14.422781.

[38] Jaclyn Smith et al. *Scalable Querying of Nested Data*. 2020. arXiv: 2011.06381 [cs.DB].

[39] Indhupriya Subramanian et al. "Multi-omics Data Integration, Interpretation, and Its Application". In: *Bioinformatics and Biology Insights* 14 (2020). PMID: 32076369, p. 1177932219899051. DOI: 10.1177/1177932219899051. eprint: https://doi.org/10.1177/1177932219899051. URL: https://doi.org/10.1177/1177932219899051.

[40] Ryan J. Urbanowicz et al. "A Rigorous Machine Learning Analysis Pipeline for Biomedical Binary Classification: Application in Pancreatic Cancer Nested Case-control Studies with Implications for Bias Assessments". In: *CoRR* abs/2008.12829 (2020). arXiv: 2008.12829. URL: https://arxiv.org/abs/2008.12829.

[41] *Expression Atlas - Fpkm*. Jan. 2021. URL: https://www.ebi.ac.uk/gxa/FAQ.html.

[42] Elizabeth Martínez-Pérez, Miguel Angel Molina-Vila, and Cristina Marino-Buslje. "Panels and models for accurate prediction of tumor mutation burden in tumor samples". In: *npj Precision Oncology* 5.1 (Apr. 2021), p. 31. ISSN: 2397-768X. DOI: 10.1038/s41698-021-00169-0. URL: https://doi.org/10.1038/s41698-021-00169-0.

[43] *One in ten rule*. Feb. 2021. URL: `https://en.wikipedia.org/wiki/One_in_ten_rule`.

[44] Leonhard Spiegelberg et al. "Tuplex: Data Science in Python at Native Code Speed". In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD/PODS '21. Virtual Event, China: Association for Computing Machinery, 2021, pp. 1718–1731. ISBN: 9781450383431. DOI: `10.1145/3448016.3457244`. URL: `https://doi.org/10.1145/3448016.3457244`.

[45] Juexin Wang et al. "scGNN is a novel graph neural network framework for single-cell RNA-Seq analyses". In: *Nature Communications* 12.1 (Mar. 2021), p. 1882. ISSN: 2041-1723. DOI: `10.1038/s41467-021-22197-x`. URL: `https://doi.org/10.1038/s41467-021-22197-x`.

[46] *Feature selection*. URL: `https://scikit-learn.org/stable/modules/feature_selection.html`.

[47] *pandas dataframe pivot*. `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.pivot.html`. Accessed: 2021-08-19.

[48] *TraNCE Github UDF branch*. URL: `https://github.com/jacmarjorie/trance/tree/udf_test/compiler/udfs`.

[49] *Transforming Nested Collections Efficiently: a framework for processing nested collection queries*. https://github.com/jacmarjorie/trance.

[50] *Transforming Nested Collections Efficiently: a framework for processing nested collection queries*. `https://github.com/jacmarjorie/trance/blob/udf_test/compiler/src/main/scala/framework/optimize/Optimizer.scala`.

[51] *UK biobank*. `https://www.ukbiobank.ac.uk/`.

[52] *Using skrebate*. URL: `https://epistasislab.github.io/scikit-rebate/using/`.

[53] *WEB-based GEne SeT AnaLysis Toolkit*. URL: `http://www.webgestalt.org/`.

Appendices

Appendix A Feature Selection Runtimes

We note that for the analysis below, the dataset was sliced randomly to output the required number of features since the features to be input to the feature selection methods did not matter, as we were merely testing the runtime of the filter. We now present the results of the experiment performed for all feature selection methods with a varying number of features and their associated runtimes, where s denotes seconds and m denotes minutes:

Table 14: **Feature selection methods runtimes**

Method	**55000**	**40000**	**30000**	**20000**	**10000**	**5000**
Chi-square	3 s	2 s	2 s	1 s	< 1 s	< 1s
ANOVA	5 s	3 s	2 s	1 s	< 1 s	< 1s
RFE	> 25 m	> 25 m	20 m s	10 m	60 s	10 s
MI	5.5 m	3.5 m	2.5 m	1.4 m	1 m	20 s
MultiSURF	7 m	4 m	3 m	2.6 m	2.1 m	1.1 m

We can see that after inputting around 35000 features to RFE, the runtime exceeds our accepted time (25 minutes), and in many occasions the Zeppelin interpreter crashes. This meant that for any queries outputting more than 35000 features, a local filter had to be pushed to filter out some features otherwise RFE could not have been used. Chi-square and ANOVA ran almost instantly regardless of the number of features input. MI's and MultiSURF's runtimes were higher, however, did not exceed 7 minutes even when using a big number of features.

Appendix B Optimizations for Multi-omics Queries

When writing programs in NRC, it is important to make sure they are optimized so that when they are executed in the generated code in the Zeppelin notebooks, they run as fast as possible. We note improvements in defining a sequential set of programs that work in pipeline fashion, which are particularly important for multi-omics programs. Here, we can build up intermediate feature matrices based on one or more datasets and then integrate others downstream. This proved beneficial when aggregating multiple datasets, i.e. build up a feature matrix with one dataset before integrating the next. These intermediate matrices reduce the cost of performing multi-modal aggregation. The GMB program defined below, which uses the data sources from section 2.2, executes the integration of the data sources as one single program. This program took approximately 3 minutes to run:

```
GMB <=
    for g in genemap union
        {(gene_name := g.g_gene_name, burdens :=
             (for o in occurrences union
                for s in clinical union
                  for e in expression union
                    for s in samples union
                     if (o.donorId = c.bcr_patient_uuid &&
                         s.bcr_patient_uuid = c.bcr_patient_uuid
                         && e.ge_aliquot = s.bcr_aliquot_uuid) then
                        for t in o.transcript_consequences union
                          if (g.g_gene_id = t.gene_id) then
                            {(sid := o.donorId,
                              lbl := if (s.gleason_pattern_primary = 2)
                                  then 0
                                     else if (s.gleason_pattern_primary =
                                         3) then 0
                                    else if (s.gleason_pattern_primary = 4)
                                        then 1
                                    else if (s.gleason_pattern_primary = 5)
                                        then 1
                                    else -1,
                               burden := (e.ge_fpkm + 0.001) *
                                    if (t.impact = "HIGH") then 0.80
                                    else if (t.impact = "MODERATE") then
                                        0.50
                                    else if (t.impact = "LOW") then 0.30
                                    else 0.01
                          )}
              ).sumBy({sid, lbl}, {burden})
            )};
```

While the program below, was executed in 45 seconds:

```
mapExpression <=
         for s in samples union
           for e in expression union
             if (s.bcr_aliquot_uuid = e.ge_aliquot) then
               {(sid := s.bcr_patient_uuid, gene := e.ge_gene_id, fpkm
                  := e.ge_fpkm)};

impactGMB <=
         for g in genemap union
           {(gene_name := g.g_gene_name, gene_id:= g.g_gene_id, burdens
               :=
             (for o in occurrences union
               for s in clinical union
                 if (o.donorId = s.bcr_patient_uuid) then
                   for t in o.transcript_consequences union
                     if (g.g_gene_id = t.gene_id) then
                        {(sid := o.donorId,
                          lbl := if (s.gleason_pattern_primary = 2) then 0
                           else if (s.gleason_pattern_primary = 3) then 0
                           else if (s.gleason_pattern_primary = 4) then 1
                           else if (s.gleason_pattern_primary = 5) then 1
                           else -1,
                          burden := if (t.impact = "HIGH") then 0.80
                                               else if (t.impact =
                                                   "MODERATE") then 0.50
                                               else if (t.impact =
                                                   "LOW") then 0.30
                                               else 0.01
                        )}
             ).sumBy({sid, lbl}, {burden})
           )};

GMB <=
         for g in impactGMB union
           {(gene_name := g.gene_name, gene_id := g.gene_id, burdens :=
             (for b in g.burdens union
               for e in mapExpression union
                 if (b.sid = e.sid && g.gene_id = e.gene) then
                   {(sid := b.sid, lbl := b.lbl, burden :=
                      b.burden*e.fpkm)}).sumBy({sid,lbl}, {burden})
                    )}
```

While the two programs above result in exactly the same output, the execution time of the first one is much longer than the second one because of the way we join the data sources.

In the first program, we simply join everything on the top level. In the second program, we first join `samples` and `gene_expression`, we then join `genemap`, `occurrences` and `clinical` and then we execute the final `sumBy` function. From the above data sources mentioned, `gene_expression` is the largest. `mapExpression` outputs results only for `gene_expression` tuples that meet the join condition with `samples` and hence there is no need to iterate over all tuples of `gene_expression`. The execution time (compared to the first program) is further reduced by the presence of the `impactGMB` since in the final program, only tuples with common id's from `occurrences` and `clinical` are used.

Appendix C Presence of Prostate Cancer Prediction

For this experiment, we used a subset of the TCGA (See section 2.1) and selected the 6 most abundant cancer types, which were stomach, bladder, esophagus & liver, pancreas, and prostate. The experiment was performed to predict the presence of prostate cancer in the samples. The experimental setup for this experiment was the same as the ones described in section 5.4.1. Initially, we had 57 000 features and ran the experiment with ANOVA and MI, since RFE's runtime was too long. We then performed the *value* filter, discussed in section 4.2. The filter filtered out any features with burden < 0.02, where 15 000 features survived. We present the results in the table below:

Table 15: **Binary classification - Prostate cancer presence prediction - value filter**

Method	**57000**	**15000**
ANOVA	79.1%	72.8%
RFE	- %	**84.0%**
MI	80.9%	76.1%

We can see that the best performing model was RFE_15000 with an accuracy of 84.0%.

We then pushed the chi-square feature-selection based filter, in order to further investigate how our models perform with decreasing number of features. We present the results in the table below:

Table 16: **Binary classification - Prostate cancer presence prediction - chi-square feature-selection based filter**

Method	**5500**	**2000**
ANOVA	80.2%	80.6%
RFE	**87.8**%	80.2 %
MI	78.7%	71.2%

We can see that indeed pushing the chi-square filter and hence reducing the feature space improved the performance of the best performing model. RFE was the best performing model, using 5500 features, outputting an accuracy of **87.8%**.

Appendix D Gene Enrichment Analysis

Additional results from the gene enrichment analysis. Figure D.1 shows the biological processes, cellular components, and molecular function categories for the 200 genes from the RFE model from the binary classification task from section 5.3.3. Figure D.2 outputs the same results but for 148 genes from RFE, excluding the genes that overlap with the output genes of the ANOVA model from the same experiment.

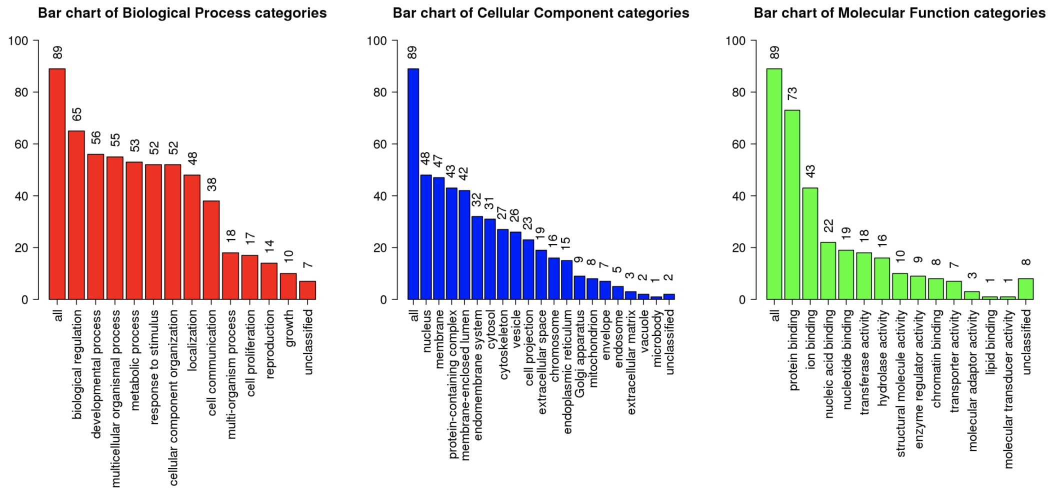


Figure D.1: Represented biological processes, cellular components, and molecular function categories for 200 genes from RFE from the binary classification - Prostate cancer severity task.

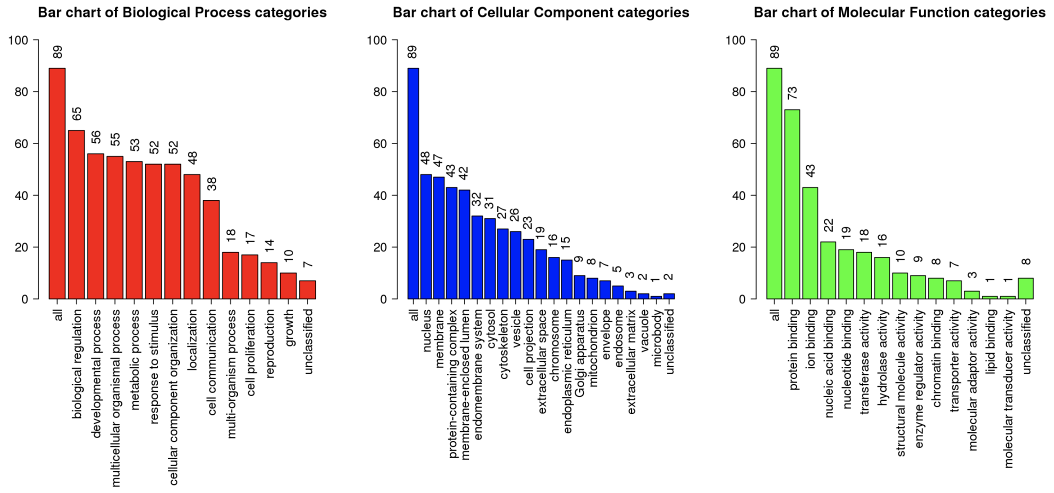


Figure D.2: Represented biological processes, cellular components, and molecular function categories for 148 genes identified from RFE only from the binary classification - Prostate cancer severity task.