Compana documentation

related to IsoQuant project

Notes by Heidi Holappa

Summer traineeship

Faculty of Science
Helsinki University
Finland

Contents

1 Introduction

This document details my participation in the research project on IsoQuant-application. My task was to investigate long reads and situations in which the start or an end of an exon had a mistake in comparison to the reference transcripts. Based on the results of the investigation a feature was then implemented to predict and correct these errors.

The document has three main sections. The second section details the code written for the investigation phase. Third presents and details the code created for error prediction and the fourth section details the code implemented to IsoQuant - application.

2 Analyzing the problem

2.1 Overview

To study the problem an application called compAna was created. The purpose of compAna is to investigate long reads and what happens in locations, where the transcript provided by IsoQuant differs from the reference data, and then try to predict and correct those errors.

The application uses the following libraries to process information: pysam, gffutils and pyfaidx.

For more information see instruction manual.

2.2 Defining offset

`gffcompare` outputs a GTF-file that contains class codes for transcripts. These class codes provide information on the quality of the alignment between the transcripts and the reference annotation. One interesting aspect in the case of misalignments is the amount of the offset. This section provides a definition for offset and details how offset is calculated in this application.

2.2.1 Definitions

Let S be a transcript from transcript_model.gtf-file produced by IsoQuant and assume that S has one or more exons as children. Let the set of these exons be $E = \{e_1, \ldots, e_n\}$, $n \in \mathbb{N}$, $\mid E \mid \geq 1$. Let R be a transcript from the reference GTF-file, to which S is being compared to. Let exon children in R be $X = \{x_1, \ldots, x_m\}$.

Each exon are given as a tuple with a start index and an end index $(a, b)\, a, b \in \mathbb{N}$, $a < b$. Let $e_i \in E$ and $x_j \in X$ be an arbitrary exons and let $e_i = (a_{e_i}, b_{e_j})$ and $x_j = (a_{x_i}, b_{x_j})$.

Assumption: for a selected transcript, the exon children do not overlap. That is, for arbitrary two exons $e_i = (a_{e_i}, b_{e_i})$ and $e_k = (a_{e_k}, b_{e_k})$ $(e_i, e_k \in E)$ it always holds that either $b_{e_i} < a_{e_k}$ or $b_{e_k} < a_{e_i}$

Definition

Definition 1. For arbitrary e_i and x_j The distance $a_{e_i} - a_{x_j}$ is the offset at the start index and the distance $b_{e_i} - b_{x_j}$ is the offset at the end index. Notice that if the offset is 'to the left', the output value is negative and similarily if the offset is 'to the right', the output value is positive.

The total offset t_{e_i,x_j} for e_i and x_j is the sum of the absolute value of these two offsets

$$t_{e_i,x_j} = \left|a_{e_i} - a_{x_j}\right| + \left|b_{e_i} - b_{x_j}\right|$$

Definition

Definition 2. For any e_i the optimal match in X to be the exon x_j that satisfies the following conditions:

1. The offset for e_i compared to x_j is the smallest possible. That is for e_i and $X = [x_1, \dots x_m]$ $\min\{t_{e_i,x_1}, t_{e_i,x_2}, \dots, t_{e_i,x_m}\} = t_{e_i,x_j}$.
2. There is no other exon $e_k \in E$ for which condition one applies with x_j and $t_{e_k,x_j} < t_{e_i,x_j}$.

2.2.2 Computing offset

The basic idea in computing the offset is to have the lists of tuples E and X in ascending order. The list E is then iterated over to find the optimal matches from list X for the exons. At each index, if possible, the items in the next index are also considered for an optimal match.

Let $e_i \in E$ and $x_j \in X$ be arbitrary elements and assume $j < |X| - 1$ and $i < |E| - 1$. Additionally assume that e_i and x_j are the first exons, for which offset is not yet computed. Four possible scenarios can happen when iterating the offset pairs in set E:

1. for exons $e_1, \dots, e_{i+1}$ and exons $x_1, \dots x_{j+1}$ it holds that e_i and x_j are an optimal match
2. for exons $e_1, \dots, e_{i+1}$ and exons $x_1, \dots x_{j+1}$ exons e_i and x_{j+1} are an optimal match
3. for exons $e_1, \dots, e_{i+1}$ and exons $x_1, \dots, x_{j+1}$ exon e_{i+1} and x_j are an optimal match
4. for exons $e_1, \dots, e_{i+1}$ and exons $x_1, \dots, x_{j+1}$ exons e_{i+1} and x_{j+1} are an optimal match

2.2.3 Illustrations

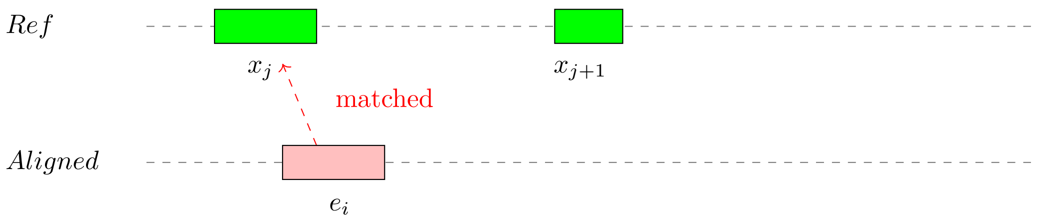

Figure 1: Offset computation: case 1

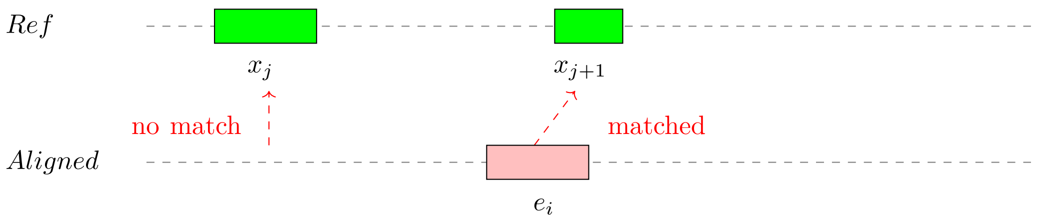

Figure 2: Offset computation: case 2

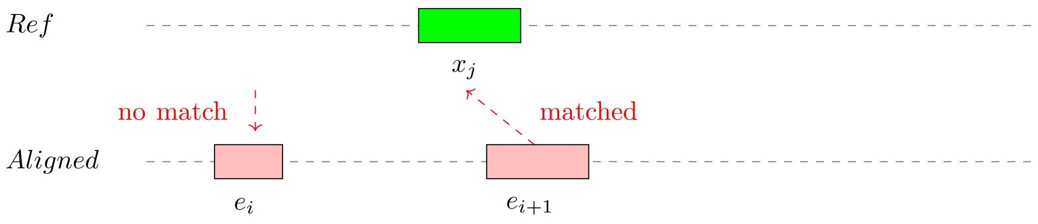

Figure 3: Offset computation: case 3

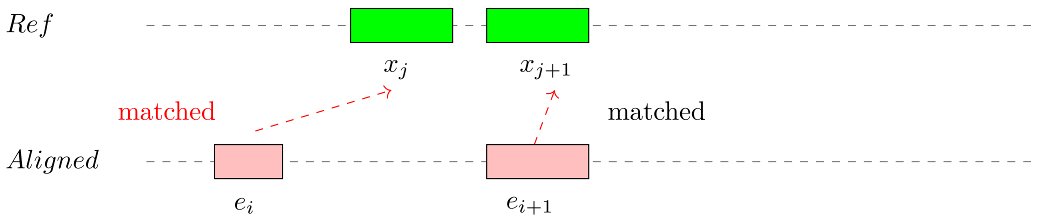

Figure 4: Offset computation: case 4

Each of these cases needs to be considered. In case of e_{i+1} being an element in the optimal match, e_{i+2} needs to be considered on the next iteration as the match could then be even more optimal.

In cases 2 and 3 an exon from either the reference data or the aligned data is left without a pair. These instances will need to be noted in the data. In case 4 e_i is paired with x_j even though for e_{i+1} x_j the offset t_{e_{i+1},x_j} was smaller, as x_{j+1} was a more suitable pair for e_{i+1}

Remark: It has not been examined in detail that the solution works in cases of longer chain of more optimal matches.

2.2.4 Pseudocode

The following pseudocode computes the offsets adhering to the rules given in definition two:

```
function compute_offset(list E, list X)
  list_of_results = []
  x_start_index = 0
  for e_index in list E:
    result = (inf, inf)
    for x_index in X from x_start_index to len(X):
      offset_between_aligned_and_reference_exon = compute total offset for E[e_index] and
          X[x_index]
      if e_index < len(E) - 1:
        offset_between_reference_exon_and_next_aligned_exon = compute total offset for
            E[e_index + 1] and X[x_index]
        if total_offset_next < total_offset:
          if x_index < len(X) - 1:
            total_offset_next_ref = compute total offset for E[e_index + 1] and X[x_index
                + 1]
            if offset_between_aligned_and_reference_exon >
                offset_between_reference_exon_and_next_aligned_exon:
              result = (inf, inf)
              break inner loop
      if x_index < len(X) - 1:
        offset_between_aligned_exon_and_next_reference_exon = compute offset E[e_index] +
            X[x_index + 1]
        if offset_between_aligned_and_reference_exon >
            offset_between_aligned_exon_and_next_reference_exon:
          if e_index < len(E) - 1:
            offset_between_next_reference_exon_and_next_aligned_exon =
                calculate_total_offset(aligned_exons[e_index+1],
                reference_exons[r_index+1])
            if not offset_between_next_reference_exon_and_next_aligned_exon <
                offset_between_aligned_exon_and_next_reference_exon:
              append (-inf, -inf) to list_of_results
              x_start_index = x_index + 1
              continue
          else:
            append (-inf, -inf) to list_of_results
            x_start_index = x_index + 1
            continue
      result = (aligned_exons[e_index][0] - reference_exons[r_index][0],
          aligned_exons[e_index][1] - reference_exons[r_index][1])
      r_start_index = r_index + 1
      break
      offset_list.append(result)
    add result to list_of_results
  if x_start_index < len(X):
    for x_index from x_start_index to len(X):
      append (-inf, -inf) to list_of_results
  return list_of_results
```

Explanation: Assume that we have a list of exons $E = [e_1, \ldots, e_n]$ and a list of reference exons $X = [x_1, \ldots, x_m]$. The algorithm starts from e_1 and iterates through the exons. Let us assume that the algorithm is now at an arbitrary index i and processing exon e_i. The following steps happen:

1. set result to $(\inf, \inf)$
2. iterate through reference exons starting from the current x_start_index to the end of the reference exons.
3. Let us remind ourselves that t_{e_i,x_j} is the total offset between e_i and arbitrary x_j. If $t_{e_i,x_j} > t_{e_{i+1},x_j}$, e_i must be $(\inf, \inf)$. Append list and break the inner loop
4. if instead $t_{e_i,x_{j+1}} < t_{e_i,x_j}$, then if $t_{e_{i+1},x_{j+1}} > t_{e_i,x_{j+1}}$, then x_j must be $(-\inf, -\inf)$. In that case append $(-\inf, -\inf)$ to the list and continue iterating through the inner loop. If the latter condition does not apply, append the offset t_{e_i,x_j} to the list and break the inner loop.
5. finally once aligned exons are iterated through if there are reference exons left, append (-inf, -inf) for each remaining reference exon to the list of results
6. once all reference exons are iterated over or the inner loop breaks, append result to the list of results.

2.2.5 Offset output

1. In case of a match the offset is expressed from the point of view of the analyzed transcript in the form of a tuple of two integers (a, b), $a, b \in \mathbb{Z}$. A negative integer indicates that the exon e_i from the analyzed data has a smaller value than the matching reference exon x_j and similarily a positive value indicates that the exon e_i has a higher value
2. A tuple $(\inf, \inf)$ indicates that no optimal match for an exon in analyzed data was found (i.e. there's possibly an exon in the analyzed data that is not present in the reference data)
3. A tuple $(-\inf, -\inf)$ that no optimal match for an exon in the reference data was found (i.e. there's possibly an exon is missing from the analyzed data)

2.3 Extracting information

2.3.1 Extracting matching cases

Once the offsets for each transcript are calculated, we can extract cases matching our interests. The offset results are iterated and for cases matching the pre-defined interesting case (wanted offset), results are extracted.

```
def extract_candidates_matching_selected_offset(offset_results: dict, offsets: tuple,
    reference_db):
  extracted_candidates = {}
  for key and value in offset_results:
    if stand is negative:
      reverse value (list)
    for offset_pair in value:
      if abs(offset_pair[0]) in the range of offsets:
        fetch the correct exon from reference_db
        location of event = exon.start + value[i][0]
        store transcript_id, location, location type as 'start', strand
      if abs(value[i][1]) == self.offset:
        fetch the correct exon from reference_db
        location of event = exon.end + value[i][0]
        store transcript_id, location, location type as 'end', strand
  return extracted_candidates
```

The values are stored as a dictionary. The key is concatenated from the transcript_id, exon number and location type (start/end) to ensure uniqueness. The dictionary contains the following values:

- **transcript_id:** transcript_id for IsoQuant transcript
- **strand:** +/-
- **exon_number:** number of exon in question (numbering starts from index 1).
- **location:** the location of the interesting event (1-based)
- **location_type:** start/end

The stored value is the location of the start or end of an exon in the IsoQuant transcript. The motivation behind this is that we want to look at what happens after the start or before the end of an exon in a pre-defined window.

2.3.2 Extracting closest canonicals

pyfaidx library provides efficient tools for extracting subsequences from a FASTA file. With these tools a window of size two times the size of the provided (or default) window is extracted from the reference FASTA file at each location stored into the matching_cases_dictionary. If the given location is marks the 'start of the exon' acceptor site canonicals are sought. The strand and the location type define the canonical bases that are sought for:

```
def iterate_matching_cases(self):
  for key, value in self.matching_cases_dict.items():
    if value["location_type"] == "start":
      splice_cite_location = value["location"] - 2
    else:
      splice_cite_location = value["location"] + 1
    coordinates = (
      key.split('.')[1],
      splice_cite_location - self.window_size,
      splice_cite_location + self.window_size
    )
    nucleotides = self.extract_characters_at_given_coordinates(coordinates)
    possible_canonicals = {
        '+': {
            'start': ['AG', 'AC'],
            'end': ['GT', 'GC', 'AT']
        },
        '-': {
            'start': ['AC', 'GC', 'AC'],
            'end': ['CT', 'GT']
        }
    }
    strand = value["strand"]
    location_type = value["location_type"]
    canonicals = possible_canonicals[strand][location_type]
    self.find_closest_canonicals(str(nucleotides), key, canonicals)
```

Method extract_character_at_given_coordinates calls pydaidx to extract a substring of nucelotides from the FASTA-file. The closest canonicals are then sought for:

```
def find_closest_canonicals(self,
  nucleotides: str,
  dict_key: str,
  canonicals: list):
  nucleotides_middle = int(len(nucleotides) / 2)
  closest_canonicals = {}
  aligned_nucleotides = nucleotides[nucleotides_middle:nucleotides_middle + 2]
  for i in range(1, nucleotides_middle):
    left_window = nucleotides[nucleotides_middle - i:nucleotides_middle - i + 2]
    right_window = nucleotides[nucleotides_middle + i:nucleotides_middle + i + 2]
    if left_window in canonicals and 'left' not in closest_canonicals:
        closest_canonicals['left'] = (left_window, aligned_nucleotides)
    if right_window in canonicals and 'right' not in closest_canonicals:
        closest_canonicals['right'] = (right_window, aligned_nucleotides)
  if left or right is missing:
    store value (aligned, nucleotides, aligned_nucleotides) to each missing key
  store results to closest_canonicals
```

2.3.3 Processing the BAM-file

Next the reads are fetched from a given BAM-file with pysam-libary. Reads are iterated through and for primary and secondary reads insertions and deletions are counted.

```
function process_bam_file(reads_and_references: dict, matching_cases: dict):
  for read in samfile.fetch():
    if read.is_supplementary:
        continue
    if read.query_name in reads_and_references:
      # validate that read has not yet been processed and read is not supplementary
      for matching_case in reads_and_references[read.query_name]:
        make correction to location (by -1)
        # validate:
        # location is in read,
        # read has a cigar string,
        # read has an end location
        # if required, add indel_errors to matching_case

        aligned_location = extract_location_from_cigar_string(
          read.cigartuples,
          read.reference_start,
          read.reference_end,
          corrected location)

        count_indels(
            read.cigartuples,
            aligned_location,
            location_type,
            strand)
        add indels to matching_case
```

2.3.4 Processing CIGAR-string

With `pysam` the cigar string for a read can be imported as a list of tuples using the `cigartuples` method. With knowing the location of the interesting event, we can now compute what cigar-codes are in the predefined window next to the location of the interesting event. To achieve this, we use the POS-information from the BAM file.

The CIGAR-parsing begins by computing the relative position:

$$\texttt{relative\_position} = \texttt{location} - \texttt{reference\_start}$$

This gives us the distance to the location of interesting event in the reference genome from the start_location, which is stored within the read (POS). At the end we are interested in the position in the CIGAR-string matching the location of the interesting event in the reference. To compute this we need to keep track of the reference position (ref_position) and simultaneously compute the aligned position.

From the SAM-tools documentation we find the following table of information:
All operation codes consume either query or reference, and the operation codes $[0, 2, 3, 7, 8]$ consume reference. As we iterate through the tuples, each tuple cumulates the aligned_pairs_position (see next pseudocode). The operation codes $[0, 2, 3, 7, 8]$ also cumulate the reference. We iterate as long as the ref_location is less than or equal to the relative_position. Once the ref_position exceeds the relative_position, we then calculate the exact aligned_pairs_position as follows.

Table 1: Operations in BAM Format

Op	BAM	Description	Consumes query	Consumes reference
M	0	alignment match	yes	yes
I	1	insertion to the reference	yes	no
D	2	deletion from the reference	no	yes
N	3	skipped region from the reference	no	yes
S	4	soft clipping	yes	no
H	5	hard clipping	no	no
P	6	padding	no	no
=	7	sequence match	yes	yes
X	8	sequence mismatch	yes	yes

Definition

Let a be the position of aligned_pairs_position, r_{ref} the ref_position, r_{rela} the relative_position and (x, n) be a cigar_tuple in which $x \in [0, 2, 3, 7, 8]$. That is x is an op code that consumes reference and n is the number of consecutive operations. Furthermore let

$$r_{\text{ref}} < r_{\text{rela}} < r_{\text{ref}} + n.$$

The final aligned position is now

$$a + n - (r_{\text{ref}} - r_{\text{rela}})$$

Some considerations:

- reads are validated before calling the method, but some validation is still done. If the given CIGAR-string does not consume any reference, -1 is returned to indicate an error has occured
- if the ref_position is points to the location of the reference_end, return the result even if ref_position is less than or equal to relative_position.

And so we get the final pseudo code:

```
function extract_location_from_cigar_string(self, cigar_tuples: list, reference_start:
     int, reference_end: int, location: int):
    relative_position = location - reference_start
    aligned_pairs_position = 0
    ref_position = 0

    for cigar_code_tuple in cigar_tuples:

        if cigar_code_tuple[0] in [0, 2, 3, 7, 8]:
            ref_position += cigar_code_tuple[1]
        if ref_position <= relative_position and not reference_start + ref_position ==
             reference_end:
            aligned_pairs_position += cigar_code_tuple[1]
        else:
            return aligned_pairs_position + (cigar_code_tuple[1] - (ref_position -
                relative_position))

    return -1
```

2.3.5 Extracting CIGAR-codes from the window next to aligned location

The method cigartuples() from pysam-library returns a list of tuples with CIGAR-codes and number of operations for each code.

The following pseudo code demonstrates how the CIGAR-codes from the given window are extracted:

```
function count_indels(cigar_tuples: list, aligned_location: int, loc_type: str):

  cigar_code_list = []
  deletions = 0
  insertions = 0
  location = 0


  if loc_type == "end":
    aligned_location = aligned_location - window_size + 1

  for cigar_code in cigar_tuples:
    if window_size == len(cigar_code_list):
      break
    if location + cigar_code[1] > aligned_location:
      overlap = location + cigar_code[1] - (aligned_location + len(cigar_code_list))
      append min(window_size - len(cigar_code_list), overlap) times cigar_code[0] to
          cigar_code_list
      location += cigar_code[1]


  for cigar_code in cigar_code_list:
    if cigar_code == 2:
      deletions += 1
    if cigar_code == 1:
      insertions += 1

  store results
```

As arguments the functions receives

- CIGAR-tuples provided by pysam
- the location of the interesting event
- the type of the location (start/end)

First a few variables are initialized. The `cigar_code_list` will contain the CIGAR op-codes in the window we are interested in. Results will be in ascending order. Variables deletions and insertions simply count indels. Location keeps track of our current aligned location. If the location type is "end", we shift the window to the 'left' side of the aligned position, taking into account that we need to do an index correction of one.

We iterate the CIGAR-tuples until we find the first event in which the sum of the current location and next CIGAR operations exceeds the aligned location. After this, we input all or as many as possible op-codes to the cigar_code_list. If the size of the list is less than the window size, we continue interating. After the window is full, we calculate the number of indels from the CIGAR-codes and save the result.

Note: At this time only deletions and insertions are counted, but the code can be easily adapted to collect information on all CIGAR-codes.

2.4 Data structures

After some benchmarking with python data structures tuple, namedtuple and dict, it appears that for this application and it's use cases, the differences in efficiency are not that notable, at least without some level of refactoring. For easier readibility and expandability dictionaries are for now used for data structures.

2.4.1 Offset results

The offset results will be returned in a dictionary of dictionaries with the following structure:

```
{
    'transcript_id': {
        'reference_id': '<str>',
        'strand': '<str>',
        'offsets': '<list of tuples>'
        'class_code': '<str>'
    }
}
```

2.4.2 Matching cases dictionary

With the offsets computated we next extract the cases of interest. These are as well stored in a dictionary of dictionaries. The key consists of the transcript_id, exon number and location to guarantee the key to be unique:

```
{
    'transcript_id.exon_<number>.loc_type': {
        'exon': '<int>',
        'location': '<int>',
        'location_type': '<str>',
        'strand': '<string>',
        'transcript_id': '<str>',
        'offset': '<int>'
    }
}
```

In later phases indel_errors and closest_canonicals are included in the matching_cases_dictionary values.

```
{
  'indel_errors': {
    'insertions': {'keys: count of insertion errors. values: count of reads'}
    'deletions': {'keys: count of deletion errors. values: count of reads'}
  }
  'closest_canonicals':{
    'left': ('closest canonical pair', 'pair in given location'),
    'right': ('closest canonical pair', 'pair in given location')
  }
}
```

For closest canonicals left and right indicate the closest pair at the given direction. If the closest pair equals to the pair in the given location, it can either mean that there is no canonical pair in the window at the given direction, or that the closest pair has the same nucleotides.

2.4.3 Transcripts and reads

Each transcript_id has one or several reads assigned to it. Same read can be assigned to multiple transcript_ids. To compute reads and locations first a dictionary of transcript_ids and read_ids is created:

```
{
  'transcript_id': { 'set of read_ids' }
}
```

Note: An assumption is made that one read is assigned to one transcript_id no more than once.

2.4.4 Reads and references

For computing the indels in given locations for each read, we finally need a list of reads and related information. Each read can be aligned to multiple transcripts.

```
{
    'read_id': {'set of matching_cases_dictionar keys'}
}
```

2.4.5 Normalizing results

After extraction and processing, the computated results for insertions and deletions are stored in a dictionary:

```
{
    ('insertion/deletion', 'strand', 'start/end', 'offset'): {'error_length <int>':
        'count <int>'}
}
```

These are output to the stdout.

Similarily for the 'closest-canonicals' -results:

```
{
  ('strand', 'exon location (start/end)', 'offset: int', 'location (left/right)'):
    {
      'key: (closest canonical, aligned pair)': {'distance <int>': 'count <int>'}
    }
}
```

As each given offset can have a positive or negative value as defined in the secion offsets, this means that for every offset in given range we have $2^4 = 16$ possible key combinations. These results are shown in the stdout and images are drawn from each key-value pair. In the stdout the results are shown as n-values. In the images the results are normalized with the following reasoning:

- transcript_ids are input into a set from the ‘matching_cases_dictionary‘
- for each transcript_id in the set reads assigned to the given transcript_id are extracted from the IsoQuant ‘model_reads.tsv‘ - output file.
- a dictionary ‘reads_and_locations‘ is created in which all cases related to a transcript_id that is related to the given read are appended into a list of locations. This means that for a given read if the read is assigned to multiple transcripts, the same location can be in the list of locations multiple times.

- reads are then imported from the BAM-file using pysam-library and for each imported read each stored location is processed (see section "extracting information" for details) and number of errors (indels) in each read are counted

After this the normalization is performed by computing the portion of values for each amount of errors:

```
normalized_values = {key: value / sum(value.values()) for key, value in
    original_values_as_dict.values()}
```

Remarks:

- each read can contain both insertions and deletions.
- if read contains multiple deletions, the given output sums separate deletions together. For instance, given a window size ≥ 8, CIGAR-string 2D3M3D would give output five deletions and zero insertions.
- for each normalized result the total number of cases is the total number of interesting locations assigned to various reads.

2.5 compAna output files

compAna outputs log-files and images. User can also option to output additional log-files for debugging. By default the output files are created into directory `<root>/logs/<datetime>/` As a default the following files are created:

- fasta_overview.md: contains information on the closest canonicals
- stdout written to a file
- a normalized graph for each combination of indel-type, strand, exon-location (left-/right side intron location) and offset.

By opting to have extended debugging on, the additional files will be created. See section "data structures" for details:

- offset-results.log
- alignment_errors.log: contains information on cases in which the whole defined or default window was filled with either insertions or deletions. Note that this is not necessarily an error.
- reads_and_location.log
- dict_of_transcripts_and_reads.log

Indel error lengths are stored in a dictionary:

```
{
    ('insertion/deletion', 'strand', 'start/end', 'offset'): {'error_length <int>':
        '<int>'}
}
```

For each key-value pair a histogram is generated.

Closest canonicals are stored in a dictionary data structure:

```
{
  ('strand', 'exon location (start/end)', 'offset: int', 'location (left/right)'): {'key:
      (closest canonical, aligned pair), value: count of instances'}
}
```

2.6 Pipeline

The pipeline gives a rough overview of the functionality of compAna-tool:

- **Initialize databases:** First library gffutils is used to create databases from the GTF-files produced by IsoQuant and gffcompare
- **Compute offset:** using the databases offsets are computed for the specified class codes (see section data structures, offsets results)
- **Extract cases:** Based on the offset results, cases within the given offset range are extracted (see section data structures, matching cases dictionary)
- **Closest canonicals:** Once a key for storing error information has been formed, the same key is used to store information on closest possible splice site. The closest possible splice site is searched from the reference fasta.
- **Compute reads and locations:** Using the model_reads.tsv produced by IsoQuant read ids are extracted (see section data structures, reads and locations).
- **Iterate reads:** Using pysam-library the reads in the provided BAM-file are iterated through and matching reads are processed.
- **Compute errors:** From matching reads the 'insertions' and 'deletions' are counted and results are stored.
- **Save point:** After computation is done the matching_case_dict is exported using pickle library. In future runs the first pipeline can be skipped, if a save file exists.
- **Output logs and graphs:** Finally the log-files and graphs are output.

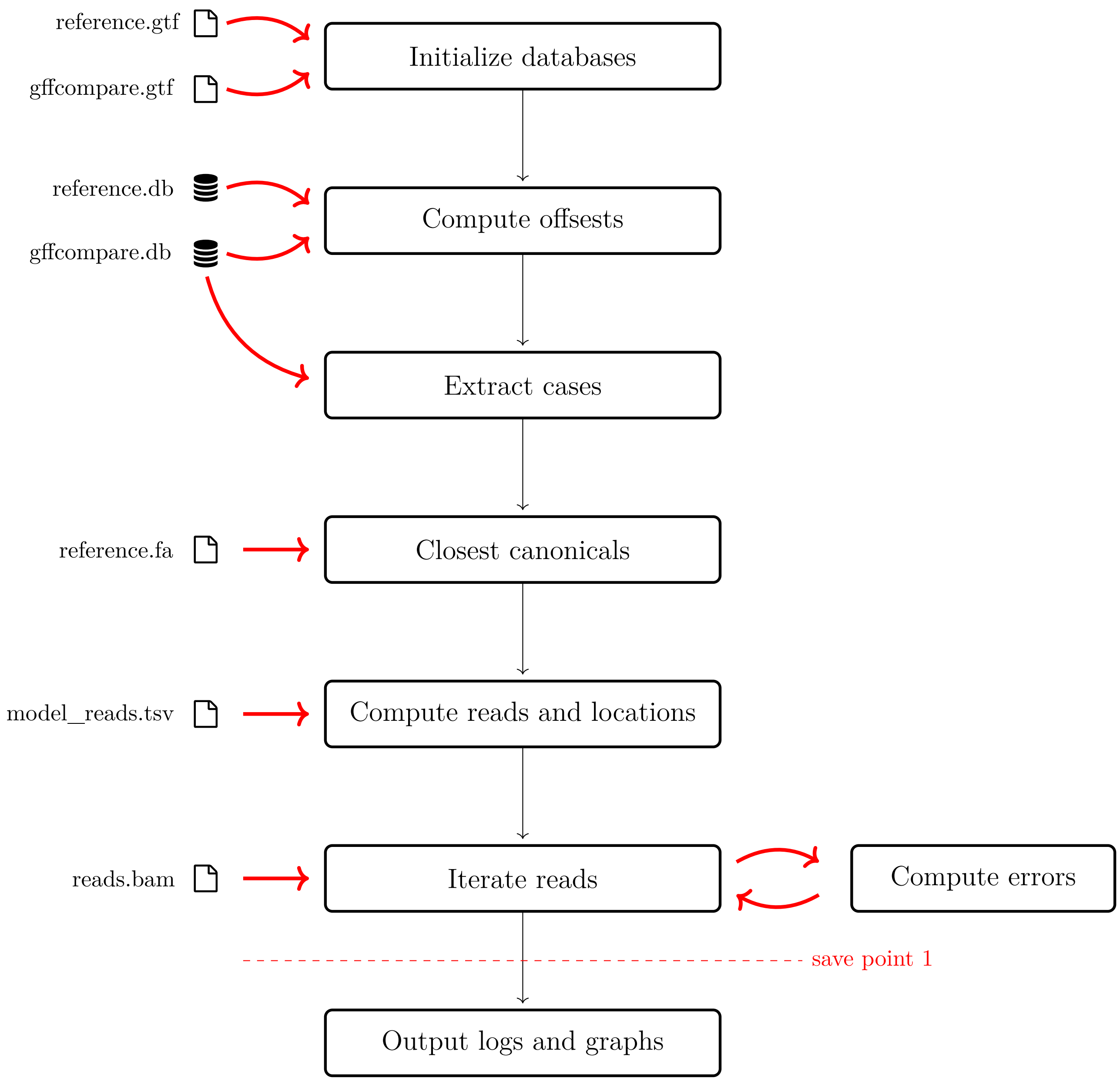

Figure 5: compAna-pipeline

3 Predicting and correcting errors

3.1 Initial plan

With the information collected in the analysis-phase it is now possible to try to implement features that can predict and correct errors in transcript. When the implementation is finished, the final output will be a corrected transcript. Precision is key. The goal is to detect and repair errors and improve precision. Ideally no new errors are created while making corrections.

As a starting point the following steps were implemented:

1. Extract data from IsoQuant transcript ('transcript_model.gtf') and IsoQuant 'model_reads.tsv' files

2. Access reads provided to IsoQuant and count indels for every intron site (=before start of the exon position and after the end of the exon position)

3. For interesting cases count closest canonical nucleotides

3.2 Pipeline for implementation

1. **Extract cases:** similarily as before the possible cases are extracted into a dictionary. This time locations and types of every intron site are extracted.

2. **Compute reads and locations:** After interesting cases and related transcripts are extracted, reads and locations are created.

3. **Compute indels:** Next for each given location insertions and deletions are computed.

4. **Closest canonicals:** For cases with interesting amounts of insertions or deletions closest canonicals are extracted from reference.fa.

5. **Make prediction:** Predictions can be made using three different strategies. See subsection 'make prediction' for more detailed summary.

5.5 **Save point:** After computation is done the intron_site_dict is exported using pickle library. In future runs the first sections of second pipeline can be skipped, if a save file exists.

6. In some way at this point it should be verified whether the results are desired or not. As a first step it would perhaps be beneficial to compare the results to the offset computations to see, how the predictions worked out.

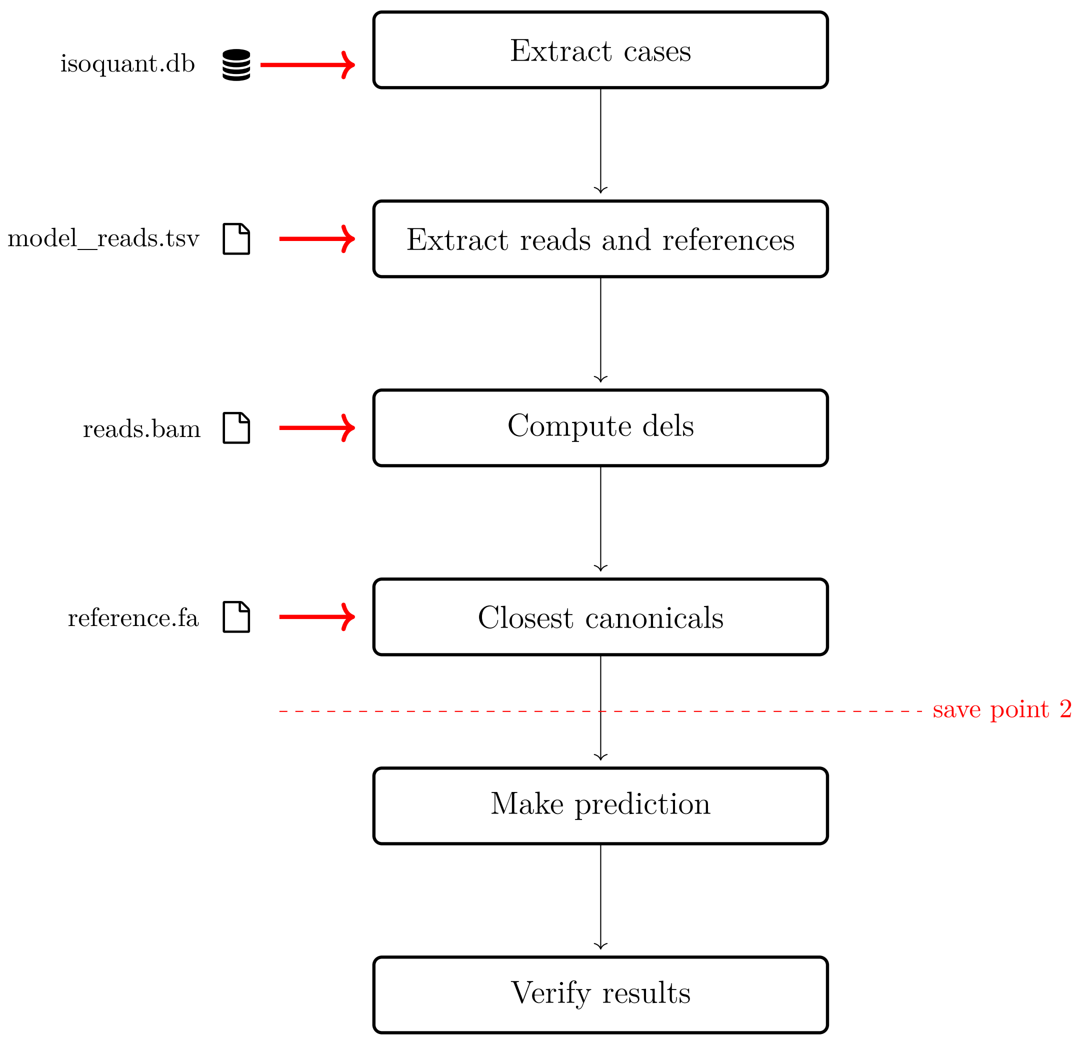

Figure 6: implementation pipeline

3.3 Extracting intron sites and information

The intron site dictionary is similar to the matching_cases_dictionary. In this case insertions, deletions and closest canonicals are extracted for both 'directions', to the left and right from the interesting location (intron site). Additionally the distribution of insertions and deletions is stored.

During prediction mean and standard deviation are computed for insertions and deletions. Based on analysis of the early results an additional data field for "most commmon del nucleotides" was included. This field contains a pair of nucleotides that resides at the distance of the most common offset case from the splice site.

```
{
  'transcript_id.location': {
    'transcript_id': <str>,
    'strand': <str>,
    'location_type': <str>,
    "location": <int>,
    "extracted_information": {
      "left": {
        "insertions": <dict>,
        "insertion_distribution": <list>,
        "insertion_mean": <float>,
        "insertion_sd": <float>,
        "deletions": <dict>,
        "deletion_distribution": <list>,
        "deletion_mean": <float>,
        "deletion_sd": <float>,
        "closest_canonical": <str>,
        "most_common_case_nucleotides": <str>
      },
      "right": {
        "insertions": <dict>,
        "insertion_distribution": <list>,
        "insertion_mean": <float>,
        "insertion_sd": <float>,
        "deletions": <dict>,
        "deletion_distribution": <list>,
        "deletion_mean": <float>,
        "deletion_sd": <float>,
        "closest_canonical": <str>,
        "most_common_case_nucleotides": <str>
      }
    }
  }
}
```

Note: In the actual implementation some of the included information may be redundant.

3.4 Computing indels

Insertions and deletions are computed similarly as in preceding phase. Differences are that in this case indels are counted for 'both directions' in each case and stored in a slightly more complex data structure. Also this time a reference to the dictionary containing the related results is passed to the method.

Info

As dictionaries in Python are mutable, only a reference to the dictionary is passed to the method counting indels.

3.5 Closest Canonicals

For each intron site closest canonicals are computed into each direction. Canonicals are then stored into the intron_site_dictionary.

3.6 Make prediction

For prediction three different strategies were experimented with.

Aggressive

1. There has to be a distinct most common case of deletions
2. A constant preset threshold has to be exceeded
3. There has to be n adjacent nucleotides that have larger or equal values to nucleotides in other positions.

The condition three may require more detailed definition.

Definition

Let S be the list of elements and $A = \{k_1, \ldots, k_n\}$ be n adjacent indices that is a sublist of S. Let $B = h_1, \ldots, h_m$ be the sublist of the remaining (possibly non-adjacent) indices in S, so that $\forall h_i \in B \; h_i \notin A$, $\forall k_j \in A \; k_j \notin B$ and $|A| + |B| = |S|$.

Now for condition 3 to apply it holds that

$$\forall S[k_j] \quad \nexists S[h_i] \text{ s.t. } S[k_j] < S[h_i].$$

Note: as this is a list of elements, it may have multiple elements with equal value.

Conservative:

1. There has to be a distinct most common case of deletions
2. There has to be a canonical pair at the distance of the most common case of deletions from the splice site

Very conservative:

1. There has to be a distinct most common case of deletions

2. There has to be a canonical pair at the distance of the most common case of deletions from the splice site
3. A constant preset threshold has to be exceeded
4. There has to be n adjacent nucleotides that have larger or equal values to nucleotides in other positions (see explanation above)

3.7 Verifying prediction

Once predicted errors have been flagged, prediction is then compared to the 'correct answers'. Results are then divided into three categories:

- **True positive:** an error was correctly predicted
- **False positive:** an error was incorrectly predicted
- **Unverified:** the case was not in the gffcompare results and can not be verified.

3.8 More efficient analysis

To chain runs with different JSON-configurations and all available strategies the following script can be used to

- run selected datasets with different startegies
- pipeline runs on multiple json-configurations
- scrape useful information from all runs to log files

Created script:

```
#!/bin/bash
rm '<stdout-history-log>'

while read F ; do
       python3 '<realpath for compana.py>' -j $F -v
       python3 '<realpath for compana.py>' -j $F
       python3 '<realpath for compana.py>' -j $F -n
done < '<realpath for json-files in txt>'


find '<string identifies for log directories>' -name '<log-file for false positives>'
    -exec sh -c 'echo "$1"; grep "" "$1"' sh {} \; > '<output-file for false-positives>'
grep "==============================================\| positives\|Unverified
    \|Verified \|Predicted \|JSON" '<stdout-history-log>' > '<output for stdout
    extraction>'

rm '<stdout-history-log>'
```

4 Implementing code to IsoQuant

4.1 General structure

The architecture of the implementation follows the same principles as the pipeline.

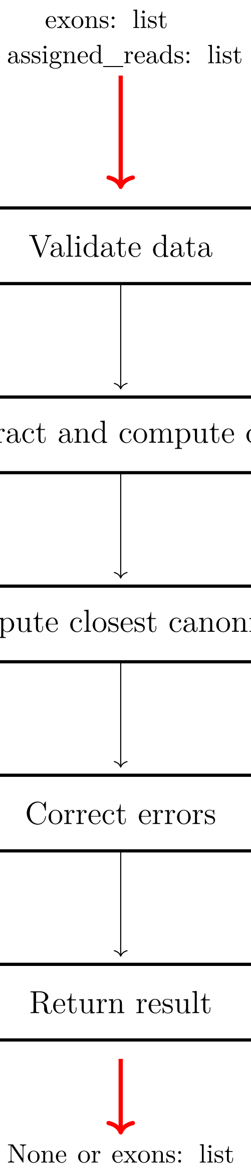

Figure 7: IsoQuant implementation

- **Validate data:** Data validation includes verifying:
  - Strand is one of the supported strands (at the time of writing either '+' or '-').
  - Read has cigartuples
- **Extract cases and count deletions:** The list assigned_reads is iterated and for each read the deletions in the given window are calculated
- **Compute closest canonicals:** For cases that fulfill all other set conditions, the closest canonicals are extracted. See section error prediction strategies for more details.

- **Correct errors:** For locations with errors the list of exons is corrected. If bases at the distance of the most common deletion are a candidate for a canonical pairing, an error is predicted to be found.
- **Return results:** If errors were found, a corrected list of exons is returned. Otherwise None is returned.

4.2 Constants

The constants are collected in the start of the function correct_transcript_splice_sites from which the code execution starts. This way they can be conveniently moved outside of the function or re-configured in the future, if needed. One possible use case would be to give the user the option to select a strategy to use, or alter the constants with arguments.

```
ACCEPTED_DEL_CASES = [3, 4, 5, 6]
SUPPORTED_STRANDS = ['+', '-']
THRESHOLD_CASES_AT_LOCATION = 0.7
MIN_N_OF_ALIGNED_READS = 5
WINDOW_SIZE = 8
MORE_CONSERVATIVE_STRATEGY = False
```

Note: Constant "Threshold cases at location" is only used when "More conservative strategy" is True. See section "error prediction strategies" for additional information.

4.3 Extracting cases and computing deletions

The assigned_reads list contains ReadAssignment objects. From each read start and end locations and cigartuples are extracted and for each splice site between the start and end location deletions are counted. First the locations within start and end of read are extracted from the exons-list. It is important to note that a read my start and end in the middle of an exon.

```
for read_assignment in assigned_reads:
    read_start = read_assignment.corrected_exons[0][0]
    read_end = read_assignment.corrected_exons[-1][1]
    cigartuples = read_assignment.cigartuples
    if not cigartuples:
        continue
    count_deletions_for_splice_site_locations(arguments)
```

For each matching location the location is first added to the splice_cite_cases dictionary, if missing. After this the deletions are computed from the cigartuples.

Info

The key-value 'del_pos_distr' is only needed for the more conservative strategy.

```
def count_deletions_for_splice_site_locations(arguments):

    matching_locations = extract_splice_site_locations_within_aligned_read(read_start,
        read_end, exons)

    for location in matching_locations:
        if location not in splice_site_cases:
            splice_site_cases[location] = {
                'location_is_end': location_type,
                'deletions': {},
                'del_pos_distr': [0 for _ in range(WINDOW_SIZE)],
                'most_common_del': -1,
                'canonical_bases_found': False
            }

        aligned_location = extract_location_from_cigar_string(arguments)

        count_deletions_from_cigar_codes_in_given_window(arguments)
```

The data structure of information to be extracted is the following:

```
{
    'location':
        {
            'location_is_end': bool,
            'deletions': dict,
            'del_pos_distr': list,
            'most_common_del': int,
            'canonical_bases_found': bool
        }
}
```

- **location_is_end:** A boolean indicating whether the location is the end of an exon
- **deletions:** A dictionary with number of deletions as key and count of reads as value
- **del_pos_distr:** A list of integers with length of the predefined window. Each index contains the count of deletions in the respective position in the window.
- **most_common_del:** Integer presenting the most common case of deletion. If no distinct case is found, the value is -1. The value also indicates direction. If the location is the start of an exon, the value is positive. Otherwise the value is negative.
- **canonical_bases_found:** A boolean stating whether there exists candidate bases for a canonical pair at the distance of the most_common_del from the current location.

The computation of deletions from cigartuples happens in two steps. First the aligned location is extracted from the cigartuple:

```python
def extract_location_from_cigar_string(arguments):
    relative_position = splice_site_location - read_start
    alignment_position = 0
    ref_position = 0

    for cigar_code in cigartuples:

        if cigar_code[0] in [0, 2, 3, 7, 8]:
            ref_position += cigar_code[1]
        if ref_position <= relative_position and not \
                read_start + ref_position == read_end:
            alignment_position += cigar_code[1]
        else:
            return alignment_position + (cigar_code[1] - (ref_position -
                relative_position))

    return -1
```

After this, the cigartuple is iterated again and starting from the aligned location a length of the predefined window_size cigarcodes are extracted.

Info

This part of the code could be optimized by performing these two operations at once.

```python
def count_deletions_from_cigar_codes_in_given_window(arguments):
    count_of_deletions = 0

    cigar_code_list = []
    location = 0

    if location_is_end:
        aligned_location = aligned_location - window_size + 1

    for cigar_code in cigartuples:
        if window_size == len(cigar_code_list):
            break
        if location + cigar_code[1] > aligned_location:
            overlap = location + \
                cigar_code[1] - (aligned_location + len(cigar_code_list))
            cigar_code_list.extend(
                [cigar_code[0] for _ in range(min(window_size -
                                        len(cigar_code_list), overlap))])
        location += cigar_code[1]

    for i in range(window_size):
        if i >= len(cigar_code_list):
            break
        if cigar_code_list[i] == 2:
            count_of_deletions += 1
            splice_site_data["del_pos_distr"][i] += 1

    if count_of_deletions not in splice_site_data["deletions"]:
        splice_site_data["deletions"][count_of_deletions] = 0

    splice_site_data["deletions"][count_of_deletions] += 1
```

4.4 Correcting errors

The main function iterates through all extracted cases. If the reads aligned to the given location exceed MIN_N_OF_ALIGNED_READS, the location is verified for errors. If MORE_CONSERVATIVE_STRATEGY is selected, two additional verifications are made.

```
def correct_splice_site_errors(arguments):
    locations_with_errors = []
    for case in splice_cite_locations:

        reads = sum of reads at current location
        if reads < MIN_N_OF_ALIGNED_READS:
            continue

        compute_most_common_del_and_verify_nucleotides(arguments)

        if MORE_CONSERVATIVE_STRATEGY:
            if not sublist_largest_values_exists(arguments):
                continue
            if not threshold_for_del_cases_exceeded(argument):
                continue

        if canonical pair is found:
            locations_with_errors.append(location of case)

    return locations_with_errors
```

The most common deletion is stored to the dictionary as it is used in error correction if an error is found. It is stored containing the distance and direction. In exon start location the value is positive and in exon end location the value is negative. For this reason an absolute value is checked against ACCEPTED_DEL_CASES.

```
def compute_most_common_del_and_verify_nucleotides(
        arguments):


    # Compute most common case of deletions
    splice_site_data["most_common_del"] = compute_most_common_case_of_deletions(
        arguments)

    # Extract nucleotides from most common deletion location if it is an accepted case
    if abs(splice_site_data["most_common_del"]) in ACCEPTED_DEL_CASES:
        extract_nucleotides_from_most_common_del_location(
            arguments)
```

For locations with a suitable most common deletion case a candidate bases for a canonical pair are verified. Strand and the location of the case (start or end of exon) is taken into consideration.

Warning!

At the time it remains an open question whether the index correction is correctly set for IsoQuant. This needs to be verified.

```python
def extract_nucleotides_from_most_common_del_location(
        arguments):
    idx_correction = -1
    extraction_start = location + most_common_del + idx_correction
    extraction_end = location + most_common_del + 2 + idx_correction
    try:
        extracted_canonicals = chr_record[extraction_start:extraction_end]
    except KeyError:
        extracted_canonicals = 'XX'

    canonical_pairs = {
        '+': {
            'start': ['AG', 'AC'],
            'end': ['GT', 'GC', 'AT']
        },
        '-': {
            'start': ['AC', 'GC', 'AC'],
            'end': ['CT', 'GT']
        }
    }

    if location is end:
        possible_canonicals = canonical_pairs[strand]['end']
    else:
        possible_canonicals = canonical_pairs[strand]['start']
    if extracted_canonicals in possible_canonicals:
        splice_site_data["canonical_bases_found"] = True
```

Finally a list of corrected exons is created:

```python
def generate_updated_exon_list(arguments):
    updated_exons = []
    for exon in exons:
            updated_exon = exon
            if exon[0] in locations_with_errors:
                corrected_location = exon[0] +
                    splice_site_cases[exon[0]]["most_common_del"]
                updated_exon = (corrected_location, exon[1])
            if exon[1] in locations_with_errors:
                corrected_location = exon[1] +
                    splice_site_cases[exon[1]]["most_common_del"]
                updated_exon = (exon[0], corrected_location)
            updated_exons.append(updated_exon)
    return updated_exons
```

In more conservative strategy two additional validations are made. There has to be n adjacent nucleotides that have larger or equal values to nucleotides in other positions (see explanation in next section):

```
def sublist_largest_values_exists(lst, n):
    largest_values = set(sorted(lst, reverse=True)[:n])
    count = 0

    for num in lst:
        if num in largest_values:
            count += 1
            if count >= n:
                return True
        else:
            count = 0

    return False
```

Additionally there has to be n (not necessarily adjacent nucleotides) for which a preset threshold is exceeded. Note that because of the first additional constraint, we can be certain that in the event of return value being True, all nucleotides in the sublist of largest values also exceed this constraint. The proving of this is left as an excersice to the reader.

```
def threshold_for_del_cases_exceeded(arguments):
    total_cases = sum of deletions
    nucleotides_exceeding_treshold = 0
    for value in del_pos_distr:
        if value > total_cases * THRESHOLD_CASES_AT_LOCATION:
            nucleotides_exceeding_treshold += 1
    return bool(nucleotides_exceeding_treshold >= abs(most_common_del))
```

4.5 Error prediction strategies

Two strategies for error prediction are available:

Conservative:

1. There has to be a distinct most common case of deletions and it is one of the accepted deletion cases
   (constant ACCEPTED_DEL_CASES).
2. There has to be a canonical pair at the distance of the most common case of deletions from the splice site
   (constant WINDOW_SIZE)
3. The number of aligned reads at the given location must exceed a preset threshold (constant MIN_N_OF_ALIGNED_READS)

Very conservative:

1. There has to be a distinct most common case of deletions and it is one of the accepted deletion cases
   (constant ACCEPTED_DEL_CASES).
2. There has to be a canonical pair at the distance of the most common case of deletions from the splice site
   (constant WINDOW_SIZE)
3. The number of aligned reads at the given location must exceed a preset threshold (constant MIN_N_OF_ALIGNED_READS)
4. There has to be atleast n indeces (n is the distinct most common case of deletions), in which a threshold for deletions has to be exceeded
   (constant THRESHOLD_CASES_AT_LOCATION)
5. There has to be n adjacent nucleotides that have larger or equal values to nucleotides in other positions (see explanation below)

The condition five of **very conservative** strategy may require more detailed definition.

Definition

Let S be the list of elements and $A = \{k_1, \ldots, k_n\}$ be n adjacent indices that is a sublist of S. Let $B = h_1, \ldots, h_m$ be the sublist of the remaining (possibly non-adjacent) indices in S, so that $\forall h_i \in B \; h_i \notin A$, $\forall k_j \in A \; k_j \notin B$ and $|A| + |B| = |S|$.

Now for condition 3 to apply it holds that

$$\forall S[k_j] \;\; \nexists S[h_i] \text{ s.t. } S[k_j] < S[h_i].$$

Note: as this is a list of elements, it may have multiple elements with equal value.