Supplementary Data of manuscript "Effect of lossy compression of quality scores on SNP calling"

Mikel Hernaez, Idoia Ochoa, Rachel Goldfeder,
Euan Ashley and Tsachy Weissman

1 SNP calling pipelines in depth

In this section we explain in more detail the different pipelines used in this work to analyze the effect that lossy compression of the quality scores has on the SNP calling. As mentioned in the main manuscript, all the considered pipelines have a preprocessing step that is common to all of them. We start by showing the commands of this preprocessing step, and specifying the commands for the variant calling and filtering steps that are specific for each pipeline. Finally, we show how the true positives and false positives were extracted.

1.1 Preprocessing step:

This preprocessing step takes as input a FASTQ file and outputs a SAM/BAM file. In theory, to analyze the effect that lossy compression of quality scores has on the SNP calling, the quality scores of the FASTQ file should be replaced by those generated by a lossy compressor. However, as we will see, some of the first commands of this preprocessing step do not use the quality scores, and thus it is not necessary to replace them directly on the FASTQ

file. Next, we show the commands of the preprocessing step, as well as the point where the quality scores are extracted, compressed, and replaced by the ones generated after decompression.

The first step is to align the pair-end FASTQ files to the reference. We use *bwa mem* as the alignment program and the NCBI build 37 of the Human reference[1] as the reference genome. We have included the -M option in *bwa mem* for compatibility with Picard tools.

```
$ bwa mem -t 4 -M ref.fa read_1.fastq read_2.fastq > aln.sam
```

Then we convert the SAM file to BAM using samtools,

```
$ samtools view -b aln.sam > aln.bam
```

and sort and index the BAM file.

```
$ samtools sort aln.bam aln.sorted.bam
```

```
$ samtools index aln.sorted.bam
```

We mark of the duplicates using Picard tools [1],

```
$ java -jar MarkDuplicates.jar INPUT=aln.sorted.bam \
      OUTPUT=aln.sorted.prededup.bam METRICS_FILE=metrics.txt
```

and add the read group information.

```
$ java -jar AddOrReplaceReadGroups.jar INPUT=aln.sorted.prededup.bam \
      OUTPUT=aln.sorted.dedup.bam RGID=group1 RGLB=lib1 \
      RGPL=illumina RGPU=unit1 RGSM=sample1
```

We remove the chimeric alignments and the duplicates,

```
$ samtools view -hb -sF 0xF00 aln.sorted.dedup.bam > aln.cleaned.bam
```

extract the desire section L, and index the resulting file.

```
$ samtools view -h -b aln.cleaned.bam L > aln.sorted.L.bam
```

[1] http://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.13/

```
$ samtools index aln.sorted.L.bam
```

Note that all the aforementioned steps do not make use of the quality scores, whereas the next ones do. Thus, we extract the quality scores at this point for compression, and replace them by the reconstructed ones. Specifically, we first convert the BAM file to a SAM file.

```
$ samtools view -h  aln.cleaned.L.bam > aln.cleaned.L.sam
```

Then we extract the *Quality Scores* and the *Flag* fields from the SAM file (the python script is a modified version of the one provided by [2] and it is included with all the scripts in the repository mentioned in the paper).

```
$ python extract_QualAndFlag.py aln.cleaned.L.sam
```

Note that some of the quality score vectors presented in the SAM file correspond to the reverse of the equivalent vector found in the FASTQ file. By extracting the flag field we are able to reverse that operation, so that the quality scores that we extract are as if they were extracted from the FASTQ file. We then compress the quality scores with a lossy compressor and replace them by the reconstructed ones (we use again the flag field to reverse the necessary ones). We do that with the following script:

```
$ python ./change_QualwFlag.py $originalSam $originalFlag \
      $qual_file $newSam
```

We then convert the SAM file back to a BAM file:

```
$ samtools view -bT $HumanReference -h -o $bamFile $newSam
```

- Further preprocessing for the GATK and the *htslib.org* pipelines:

The GATK and *htslib.org* pipelines require some further steps, that we describe next. Specifically, we perform a local Indel realignment, which is performed using GATK. The commands used in this paper to perform the

local Indel realignment are:

Create the target list of intervals:

```
$ java -jar GenomeAnalysisTK.jar -T RealignerTargetCreator \
      -R pathHumanReference -L targetRegion -I bamFile \
      -known bundle_2.8/Mills_and_1000G_gold_standard.indels.b37.vcf \
      -o target_intervals_list
```

Perform Realignment:

```
$ java -jar GenomeAnalysisTK.jar -T IndelRealigner \
      -R pathHumanReference -L targetRegion -I bamFile \
      -targetIntervals target_intervals_list \
      -known bundle_2.8/Mills_and_1000G_gold_standard.indels.b37.vcf \
      -o bamFile_realign
```

Then, a recalibration of the quality scores is performed using the following two commands:

```
$ java -jar GenomeAnalysisTK.jar -T BaseRecalibrator -R pathHumanReference
      -I bamFile_realign -L targetRegion \
      -knownSites bundle_2.8/dbsnp_138.b37.vcf \
      -knownSites bundle_2.8/Mills_and_1000G_gold_standard.indels.b37.vcf \
      -knownSites bundle_2.8/1000G_phase1.indels.b37.vcf -o recal_data
```

```
$ java -jar GenomeAnalysisTK.jar -T PrintReads -R pathHumanReference \
      -I bamFile_realign -L targetRegion -BQSR recal_data -o bamFile_recal
```

This concludes the preprocessing step. The resulting file serves as input to the variant callers of each of the pipelines. The commands for these pipelines are described next.

1.2 Variant Calling and Filtering steps:

- GATK pipeline:

We use the Haplotype Caller as the variant caller for the GATK pipeline.

```
$ java -jar GenomeAnalysisTK.jar -T HaplotypeCaller -R pathHumanReference \
      -I bamFile_recal -L targetRegion --dbsnp bundle_2.8/dbsnp_138.b37.vcf
      - -genotyping_mode DISCOVERY -stand_emit_conf 10 -stand_call_conf 30
```

Once the calls are made, we extract the SNPs using:

```
$ java -jar GenomeAnalysisTK.jar -T SelectVariants -R pathHumanReference \
      -V input.vcf -L targetRegion -selectType SNP -o outputSNPS
```

We further filter the calls either using a hard filter:

```
$ java -jar GenomeAnalysisTK.jar -R pathHumanReference -T VariantFiltration
      -L targetRegion -V input.vcf --filterExpression ``filterExpression''
      --filterName ``filterName'' -o outputFilteredVCF
```

where the filter expression name is given by "QD < 2.0 || FS > 60.0 || MQ $<$ 40.0 || HaplotypeScore > 13.0 || MQRankSum < -12.5 || ReadPosRankSum < -8.0", or using the VQSR command:

```
$ java -jar GenomeAnalysisTK.jar -R $pathHumanReference \
      -T VariantRecalibrator -L $targetRegion -input input.vcf \
      -resource:$resource1 -resource:$resource2 -resource:$resource3 \
      -resource:$resource4 $recalParams -mode SNP \
      -tranche 100.0 -tranche 99.9 -tranche 99.0 -tranche 90.0 \
      -recalFile $recalFile -tranchesFile $tranchesFile -rscriptFile $rscri
      "$filterExpression" --filterName "$filterName" -o $outputFilteredVCF
```

- *htslib.org* pipeline:

We call the variants using two commands. First the *mpileup* command from Samtools, and then the *call* command from BCFtools:

```
$ samtools mpileup -ugf $pathHumanReference $bamFile_recal |
      bcftools call -vmO v -o $outputVCF
```

Once the calls are made, we extract the SNPs using:

```
$ java -jar GenomeAnalysisTK.jar -T SelectVariants -R pathHumanReference \
      -V input.vcf -L targetRegion -selectType SNP -o outputSNPS
```

After calling the SNPs the following filter is applied:

```
$ bcftools filter -O v -o $outputFilteredVCF -s $filterName  \
      -i $filterExpression $outputSNPS \
      --refFile=$pathHumanReference --output=$outputVCF
```

- Platypus pipeline:

We call the variants using the Platypus variant caller:

```
$ python Platypus_0.8.1/Platypus.py callVariants --bamFiles=$bamFile \
      --refFile=$pathHumanReference --output=$outputVCF
```

Once the calls are made, we extract the SNPs using:

```
$ java -jar GenomeAnalysisTK.jar -T SelectVariants -R pathHumanReference \
      -V input.vcf -L targetRegion -selectType SNP -o outputSNPS
```

No further filtering is performed in this pipeline.

1.3 True and False positive extraction:

Finally, the true and false positives in each of the generated VCF files are computed using the *SelectVariants* command from the GATK toolkit. For example, to generate the false positives the command would be:

```
$ java -jar GenomeAnalysisTK.jar -R $pathHumanReference \
      -T SelectVariants -L $targetRegion --variant $vcf_file \
      --discordance $goldenStandardVCF -o $outputFPVCF
```

2 INDEL calling pipelines in depth

In this section we detail the different pipelines used to analyze the effect that lossy compression of base quality scores has on the INDEL calling.

2.1 Preprocessing

First, we align paired-end FASTQ files to the reference. We use BWA mem for sequence read alignment to the human reference genome hg19 from UCSC.

```
$ bwa mem -M -R "@RG\tID:sim\tSM:sim" $ref ${file_prefix}1.fq \
      ${file_prefix}2.fq > ${file_prefix}.sam
```

Then we convert the SAM file to BAM,

```
$ samtools view -b -S ${file_prefix}.sam > ${file_prefix}.bam
```

and finally, sort, remove PCR duplicates and index the BAM file:

```
$ samtools sort ${file_prefix}.bam ${file_prefix}.sorted
```

```
$ samtools rmdup ${file_prefix}.sorted.bam \
      ${file_prefix}.sorted.rmdup.bam
```

```
$ samtools index ${file_prefix}.sorted.rmdup.bam
```

2.2 Variant Calling and Filtering Steps

- Dindel (version 1.01):
We first extract the indels from the bam file and infer library insert size distribution.

```
$ dindel --analysis getCIGARindels --bamFile ${prefix}.bam  \
      --outputFile ${prefix}.dindel_output --ref ${ref}
```

Then, make realignment windows

```
$ python makeWindows.py --inputVarFile \
      ${prefix}.dindel_output.variants.txt  --windowFilePrefix \
      --numWindowsPerFile 1000
```

to later realign all windows.

```
for a in `ls window_files_${prefix}/*`
do
      i=`echo $a | sed 's/.txt$//' | sed 's/.*realign\_windows\.//'`
```

```
$ dindel --analysis indels --doDiploid --bamFile $bam --ref ${ref} \
     --varFile window_files_${prefix}/${prefix}.realign_windows.${i}.txt \
     --libFile ${prefix}.dindel_output.libraries.txt  \
     --outputFile ${prefix}.dindel_stage2_output_windows.${i}
```

We finally integrate the results into a VCF file with the following two commands,

```
$ ls ${prefix}.dindel_stage2_output_windows.*.glf.txt >> \
       ${prefix}.dindel_stage2_outputfiles.txt
```

```
$ python mergeOutputDiploid.py \
       --inputFiles ${prefix}.dindel_stage2_outputfiles.txt \
       --outputFile ${prefix}.variantCalls.VCF --ref ${ref}
```

and filter the results

```
$ vcftools --remove-filtered hp10 --recode \
       --remove-filtered q20 --vcf ${prefix}.variantCalls.VCF \
        --out ${prefix}.dindel.filtered.vcf
```

- FreeBayes:

We run FreeBayes with the following command:

```
$ freebayes -f ${ref} ${prefix}.bam > ${prefix}.vcf
```

and extract the INDELS as:

```
$ vcftools --keep-only-indels --recode --vcf ${prefix}.vcf \
       --out ${prefix}.freebayes.filtered.vcf
```

- Preprocessing for Unified Genotyper:

First, we perform realignment around the INDELS by first creating the targets:

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
       -T RealignerTargetCreator \
       -R $ref\
       -o $sample.intervals \
       -I $sample.bam \
       --known Mills_and_1000G_gold_standard.indels.hg19.vcf \
       --known 1000G_phase1.indels.hg19.vcf
```

and then performing the actual realignment:

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
       -T IndelRealigner \
       -R $ref\
       -o $sample.realigned.bam \
       -targetIntervals $sample.intervals \
```

```
        -I $sample.bam \
        --knownAlleles Mills_and_1000G_gold_standard.indels.hg19.vcf  \
        --knownAlleles  1000G_phase1.indels.hg19.vcf
```

We then clip the reads

```
$ java -Xmx10g -jar GenomeAnalysisTK.jar \
        -T ClipReads \
        -I $sample.realigned.bam \
        -o $sample.realigned.clipped.bam \
        -R $ref \
        -CT 500-1000 \
        -CR WRITE_Q0S \
        -os $sample.clipped.txt
```

and compute Base Score Quality Recalibration (BSQR)

```
$ java -Xmx10g -jar GenomeAnalysisTK.jar \
        -T BaseRecalibrator \
        -I $sample.realigned.clipped.bam \
        -R $ref \
        -knownSites dbsnp_137.hg19.vcf \
        -knownSites Mills_and_1000G_gold_standard.indels.hg19.vcf  \
        -knownSites 1000G_phase1.indels.hg19.vcf \
        -o $sample.recal_data.grp \
        -nct 4 \
        -filterMBQ
```

We finally print the reads,

```
 $ java -jar -Xmx10g GenomeAnalysisTK.jar \
  -T PrintReads \
  -R $ref \
  -I $sample.realigned.clipped.bam \
  -BQSR $sample.recal_data.grp \
  -o $sample.final.bam
```

reduce the size for lightweight storage

```
$ java -Xmx10g -jar GenomeAnalysisTK.jar \
  -R $ref \
  -T ReduceReads \
  -I $sample.final.bam \
  -o $sample.rr.final.bam
```

and index the final bam

```
$ samtools index $sample.rr.final.bam
```

- Unified Genotyper:

After the preprocessing described above we run the Unified Genotyper variant caller as following:

```
$ java -Xmx8g -jar GenomeAnalysisTK.jar \
       -T UnifiedGenotyper \
       -R $ref \
       -I $sample.bam \
       -glm INDEL \
       -stand_call_conf 30 \
       -o $sample.UG.vcf
```

and filter the ouput:

```
$ java -Xmx8g -jar GenomeAnalysisTK.jar \
       -T VariantFiltration \
       -R $ref \
       --variant $vcf \
       --filterExpression "QD␣<␣2.0" \
       --filterName "QD" \
       --filterExpression "ReadPosRankSum␣<␣-20.0" \
       --filterName "ReadPosRankSum" \
       --filterExpression "FS␣>␣200.0" \
       --filterName "FS" \
       -o ${vcf}.filtered.vcf
```

- Preprocessing for Haplotype Caller:

First, we perform realignment around the INDELS by first creating the targets:

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
       -T RealignerTargetCreator \
       -R $ref\
       -o $sample.intervals \
       -I $sample.bam \
       --known Mills_and_1000G_gold_standard.indels.hg19.vcf \
```

and then performing the actual realignment:

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
       -T IndelRealigner \
```

```
        -R $ref\
        -o $sample.realigned.bam \
        -targetIntervals $sample.intervals \
        -I $sample.bam \
        --known Mills_and_1000G_gold_standard.indels.hg19.vcf  \
```

Next, we analyze patterns of covariation:

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
   -T BaseRecalibrator \
   -R $ref \
   -I $sample.realigned.bam \
   -knownSites $dbsnp \
   -knownSites Mills_and_1000G_gold_standard.indels.hg19.vcf \
   -o $sample.recal_data.table
```

We do a second pass to analyze covariation after recalibration

```
$ java -jar -Xmx10g GenomeAnalysisTK.jar \
   -T BaseRecalibrator \
   -R $ref \
   -I $sample.realigned.bam \
   -knownSites $dbsnp \
   -knownSites Mills_and_1000G_gold_standard.indels.hg19.vcf \
   -BQSR $sample.recal_data.table \
   -o $sample.post_recal_data.table
```

Apply recalibration to sequence data

```
$  java -jar GenomeAnalysisTK.jar \
   -T PrintReads \
   -R $ref \
   -I $sample.realigned.bam \
   -BQSR $sample.post_recal_data.table \
   -o $sample.recal_reads.bam
```

- HaplotypeCaller:

After the preprocessing described above we run the HaplotypeCaller variant caller as following:

```
$ java -Xmx8g -jar GenomeAnalysisTK.jar \
        -T HaplotypeCaller \
        -R $ref \
        -I $sample.recal_reads.bam \
        --genotyping_mode DISCOVERY \
```

```
        -stand_emit_conf 10 \
        -stand_call_conf 30 \
        -o $sample.variants.HC.vcf
```

Extract INDELs

```
$ java -jar GenomeAnalysisTK.jar \
   -T SelectVariants \
   -R $ref \
   -V $sample.variants.HC.vcf \
   -selectType INDEL \
   -o $sample.raw_indels.vcf
```

and filter the ouput:

```
$ java -Xmx8g -jar GenomeAnalysisTK.jar \
        -T VariantFiltration \
        -R $ref \
        --V $sample.raw_indels.vcf \
        -filterExpression "QD<2.0||FS>200.0||ReadPosRankSum<-20.0" \
        --filterName "bestPractices_indel_filter" \
        -o ${vcf}.filtered.vcf
```

3 Lossy Compressors

In this section we describe the commands of each of the lossy compressors, together with the parameters that we employ in each of them.

3.1 QVZ

For SNP calling, we have run QVZ for nine rates:

$$R = \{0, 0.05, 0.1, 0.2, 0.4, 0.6, 0.8, 0.9, 1\},$$

two different cluster sizes:

$$C = \{1, 3\},$$

and three different distortions. Regarding the distortions, QVZ computes the distortions as a matrix D where its position of the matrix D_{ij} is the cost

of representing i as j. It comes with three built-in distortions, namely:

$$D = \{M, A, L\},$$

where $M_{ij} = |i - j|^2$, $A_{ij} = |i - j|$ and $L_{ij} = \log(1 + |i - j|)$, which are the ones used for the analysis.

For indel calling, we have run QVZ for rates R = {0,0.3,0.7,0.9}, 3 clusters and MSE distortion. The selection of MSE distortion and 3 clusters is based on the results presented for the SNPs.

When a built-in distortion is used the command used to compress the quality scores is:

```
$ qvz -f R -c C -d D -u reconstructed.quals -s input.quals compressed.out
```

where R indicates the rate, C the number of clusters and D the distortion used.

3.2 R/Block

For SNP calling, we have run PBlock with the following values of p:

$$p = \{1, 2, 4, 8, 16, 32\}$$

and RBlock with the following values of r:

$$r = \{3, 8, 10, 11, 12, 15, 20, 25, 30\}.$$

For indel calling we have chosen the following values: p = {2,8,16} and r = {3,8, 10}.

The commands used for the compression of the quality scores are for PBlock:

```
$ CompressQualFile input.quals -q 1 -m 1 -l p
```

and for RBlock:

```
$ CompressQualFile input.quals -q 2 -m 1 -l r
```

The command for decompression is the same in both cases and it is given by:

```
$ DecompressQual compressed.cqual
```

3.3 Illumina Binning - DSRC2

In order to compress the quality scores using the binning proposed by Illumina, we choose the FASTQ file compressor DSRC2. Unfortunately, DSRC2 does not have the option of solely compressing the quality scores, thus some preprocessing of the file is needed.

We start by creating a dummy FASTQ file where the quality scores are the ones in the SAM file [reverse complemented if needed], and the identifiers and reads are all set to the same values. Since DSRC2 uses an arithmetic encoder, the identifiers and reads will not account for almost any space of the compressed file (see [3] for more details of this preprocessing). After the dummy FASTQ file is created, the following command is used for compression:

```
$ dsrc c -m2 -l input_dummy.fastq output.compressed
```

The decompression is done using the command:

```
$ dsrc d input.compressed ouptut.dsrc2.fastq
```

Finally, the quality scores are extracted from the reconstructed FASTQ file as follows:

```
$ awk 'NR\%4 == 0' input.dsrc2.fastq > output.dsrc2.qual
```

4 More Results regarding the performance on SNP calling

In this section we show in more detail the results obtained in the SNP calling pipelines when the quality scores are replaced by the reconstructed ones (after lossy compression). We first analyze the results in terms of F.P. versus T.P., for the different pipelines, datasets and ground truth. Next we discuss the values of the sensitivity, precision and f-score, which are summarized in tables. Finally, we provide the ROC curves that are generated by filtering the VCF files by a given parameter.

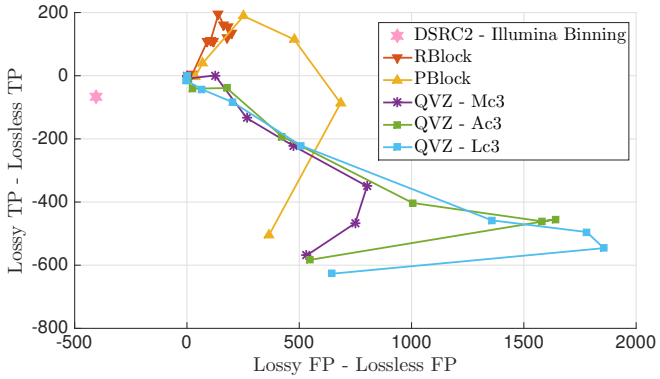
4.1 Performance in terms of F.P. and T.P.

Here we provide extra plots that show the behavior in terms of F.P. vs T.P. We normalize the number of T.P. and F.P. with those generated with the lossless (original) file, such that the number of T.P. and F.P. of the lossless VCF after normalization become 0. We then join the points that belong to a given algorithm, sorted by size, such that the point closer to (0,0) corresponds to the largest size. Thus a positive number of T.P. (F.P.) means that the corresponding VCF file contains more T.P. (F.P.) than the one generated with the original quality scores. Ideally, we would like to get points close to (0,0), meaning that the behavior is similar to that of the original quality scores; or even better, points that have a positive number of T.P. and a negative number of F.P., since that would indicate a better behavior while reducing the size of the quality scores.

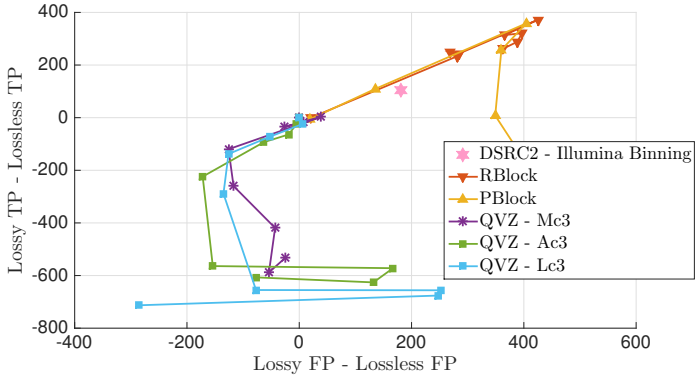
Due to the large number of simulations, we divide the section based on the datasets, namely, NA12878 15× and 30× coverage (for both chromosome 11 and 20).

1. NA12878 15× Coverage Dataset (ERR174310)

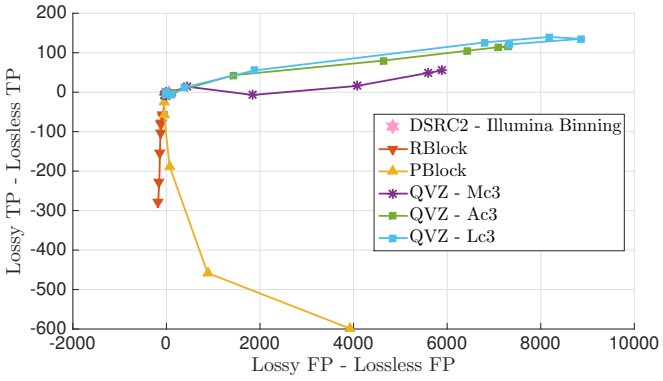
   Chromosome 11, NIST ground truth: We start by discussing the behavior of the different algorithms with the chromosome 11, when



(a) GATK pipeline



(b) *htslib.org* pipeline



(c) Platypus pipeline

Figure 1: F.P. vs T.P. for different lossy compressors when using NIST ground truth and the chromosome 11 of the NA12878 low coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.

the NIST ground truth is used for comparison. Fig. 1 shows the results for the three pipelines considered in this paper. As it can be observed, Illumina's proposed binning in GATK reduces the F.P. while maintaining a similar number of T.P. In *htslib.org* and Platypus it introduces several F.P., and some T.P. Note that in these two pipelines Illumina's proposed binning is outperformed by other lossy compressors (in the sense that for the same number of F.P., there are other points with larger number of T.P.). RBlock produces several F.P. and T.P. in both GATK and *htslib.org*, and as the rate increases it reduces the number of F.P. while maintaining a similar number of T.P. A different behavior is observed in Platypus, where both the F.P. and T.P. are negative, even when the rate increases, even though the F.P. decreases and the T.P. increases. PBlock with high rate offers a similar behavior of that of RBlock (in all three pipelines), whereas with lower rates the performance deteriorates by decreasing both the number of T.P. and F.P. Regarding QVZ, in the figure we show its performance for 3 clusters, when the distortion criteria is given by MSE (M), L1 (A) and Lorentzian (L). As it can be observed, the three distortions offer a similar performance for the three pipelines. In general, for lower rates MSE performs better, and as the rate increases there is no much difference between the three. In GATK they introduce several F.P. and decrease the number of T.P., and as the rate increases they approach the (0,0) point by reducing the number of F.P. and increasing the number of T.P. In *htslib.org* a different behavior is observed. Both the T.P. and the F.P. are decreased (except for the smallest rates), that is, fewer F.P. are called with respect to the uncompressed. On the contrary, in Platypus both the number of F.P. and T.P. is increased, even for smaller rates. The behavior of QVZ with one cluster is shown in Fig. 2. As it can be observed, for all the three pipelines, the overall behavior is like that of 3 clusters, but with more F.P. and less T.P. in general. This is true for all the simulated datasets, including the case where the Illumina ground

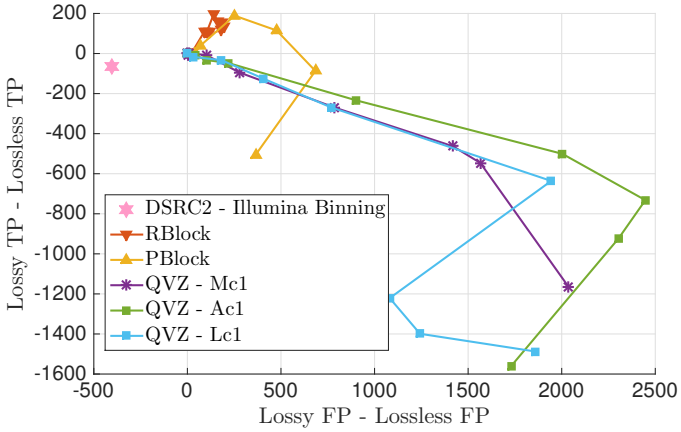
truth is used instead of the NIST. Therefore, hereafter, we omit the discussion of QVZ with one cluster and focus on that of 3 clusters.

Regarding the lossy compressors, R/PBlock offer a similar performance, which is quite different from that of QVZ. In terms of the pipelines, Platypus makes more calls in general, followed by GATK and *htslib.org* (look at the number of the axis).

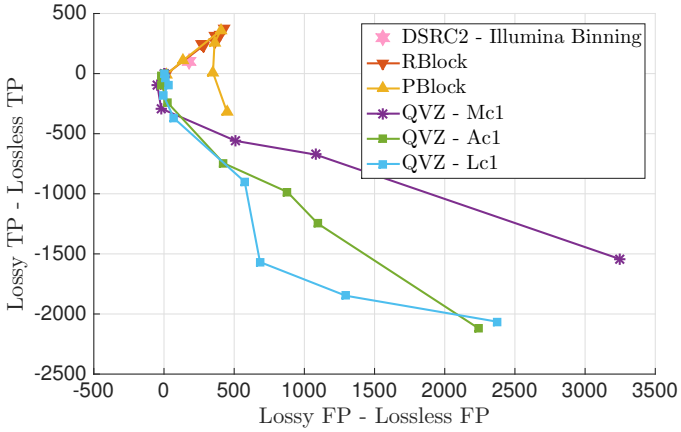
Chromosome 20, NIST ground truth: The results of the chromosome 20 for all three pipelines and NIST ground truth are shown in Fig. 3. As can be observed, the overall behavior of the different algorithms is very similar to that observed for chromosome 11. The main difference is regarding the scales of the F.P. and T.P., which is due to the fact that chromosome 20 is smaller than chromosome 11, and thus fewer calls are made in general. Also, chromosome 20 contains fewer true SNPs (see Fig. 1. of main manuscript). Worth noticing is the fact that there is a higher decrease in F.P. than in T.P. with respect to the behavior with chromosome 11. That is, the number of T.P. remains similar than with chromosome 11, but much fewer F.P. are obtained. Overall, the same conclusions can be drawn looking at these results.

Chromosome 11, Illumina ground truth:

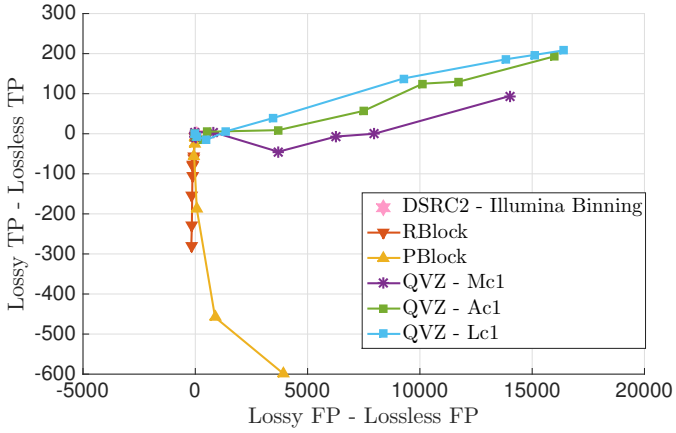
Fig. 4 shows the results of chromosome 11 when using the Illumina ground truth. Note that the VCF files generated by each pipeline are still the same as when using the NIST ground truth, the only difference is that with the Illumina ground truth the number of T.P. and F.P. may differ. For example, Illumina's proposed binning in GATK achieves a very similar performance than with the NIST ground truth, whereas it improves with *htslib.org* and Platypus (e.g., it contains more T.P. and fewer F.P.). RBlock and PBlock, on the contrary, improve both in GATK and *htslib.org*, but they worsen in Platypus. Finally, QVZ with GATK improves its performance at high rates, whereas its performance deteriorates at low rates. With *htslib.org* the performance is slightly



(a) GATK pipeline

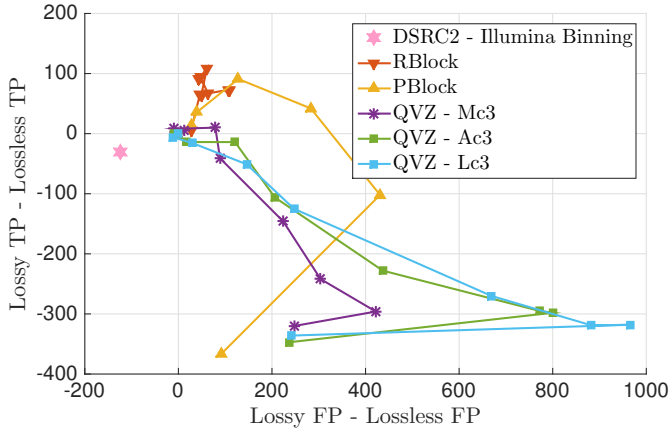


(b) *htslib.org* pipeline

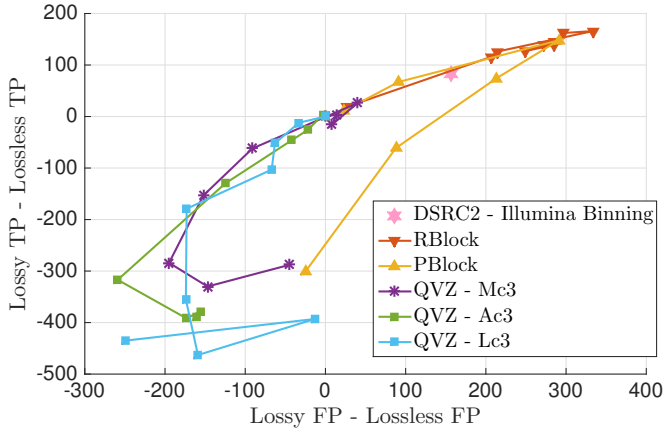


(c) Platypus pipeline

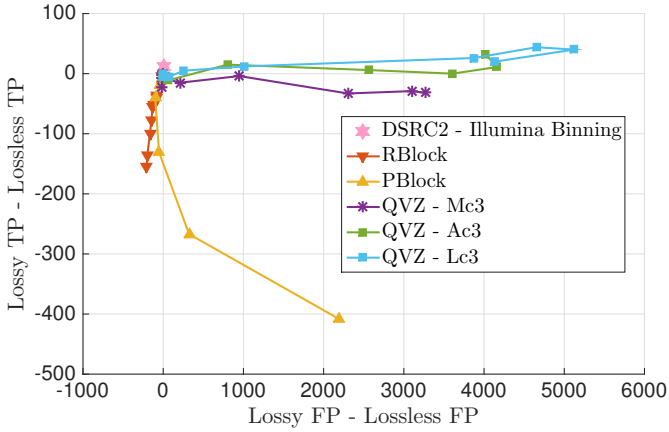
Figure 2: F.P. vs T.P. for different lossy compressors when using NIST ground truth and the chromosome 11 of the NA12878 15× coverage dataset, for the three considered pipelines. QVZ is shown with 1 cluster.



(a) GATK pipeline



(b) *htslib.org* pipeline



(c) Platypus pipeline

Figure 3: F.P. vs T.P. for different lossy compressors when using NIST ground truth and the chromosome 20 of the NA12878 15× coverage dataset, for the three considered pipelines.

worse, and in Platypus is almost identical.

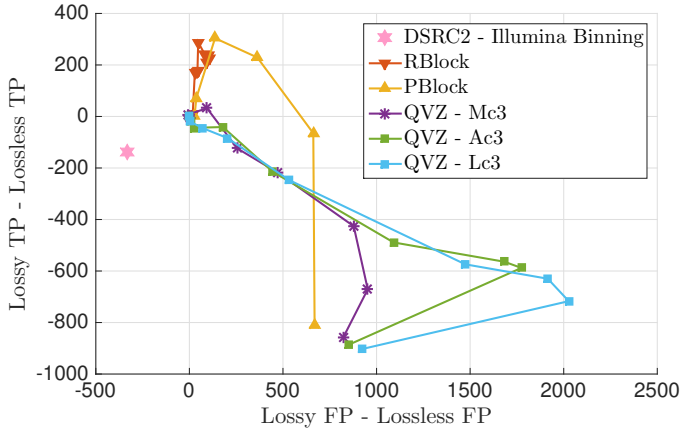
Chromosome 20, Illumina ground truth:

Fig. 5 shows the results for chromosome 20 when using the Illumina ground truth. Similarly to the previous case, the results are very similar to those obtained with the NIST ground truth. Illumina's proposed binning deteriorates its performance in GATK, remains almost identical in Platypus and improves considerably in *htslib.org*. As before, RBlock and PBlock improve both in GATK and *htslib.org*, but they worsen in Platypus. Finally, the QVZ performance improves in GATK at high rates and in *htslib.org*, and remains almost identical in Platypus.

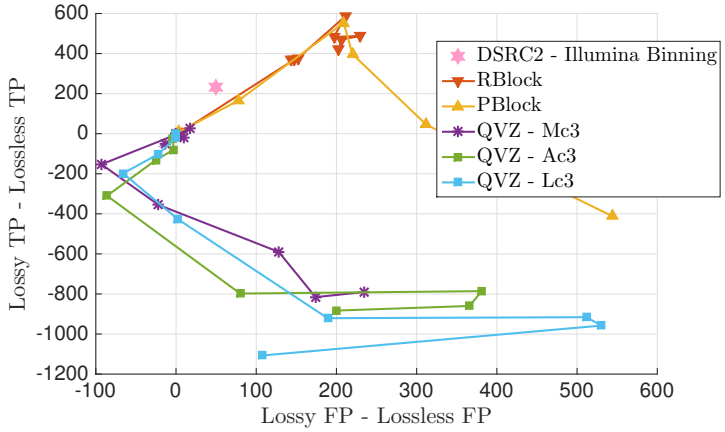
2. NA12878 30× Coverage Dataset (ERR262996)

Chromosome 11, NIST ground truth:

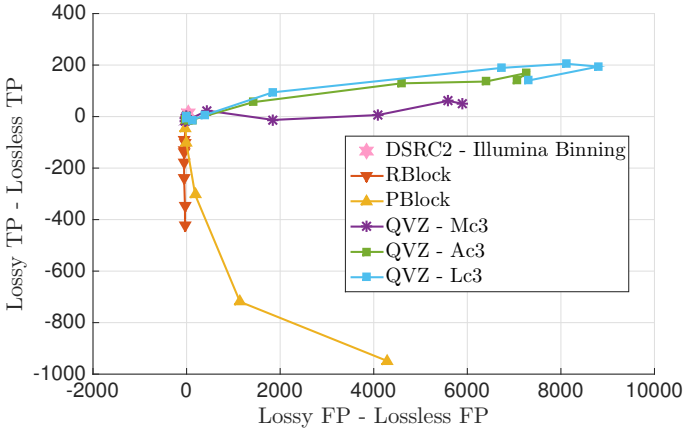
The results regarding chromosome 11 with the NIST ground truth are presented in Fig. 6. As can be observed, Illumina proposed binning achieves a performance close to that of the uncompressed one, for all the studied pipelines. Note also that in all the cases it is outperformed by other algorithms, in the sense that for the same number of F.P., they achieve more T.P.. RBlock with GATK and *Platypus* achieves more F.P. and more T.P. than in the 15× coverage dataset, whereas in Platypus it achieves fewer F.P. and T.P. Note also that in GATK increasing the rate does not necessarily yield a better performance. Finally, RBlock can not simultaneously improve with respect to the uncompressed in the F.P. and T.P. PBlock is able to achieve with GATK fewer F.P. and more T.P. than with the uncompressed. The behavior with respect to the previous dataset in *htslib.org* is similar, but with more T.P. and F.P., whereas in Platypus it gets more F.P. and fewer T.P. QVZ with GATK and small rates achieves more F.P. and T.P. than uncompressed, but as the rate increases, it outperforms it in both F.P. and T.P. Note that this is true for all three distortions (MSE, L1 and Lorentzian). In *htslib.org*, QVZ decreases the number of F.P. with



(a) GATK pipeline

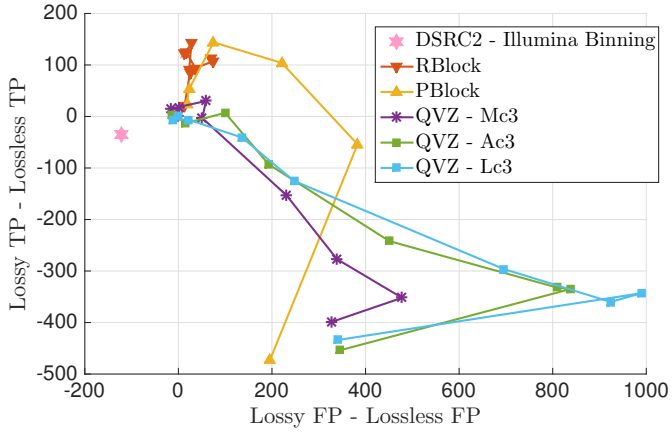


(b) *htslib.org* pipeline

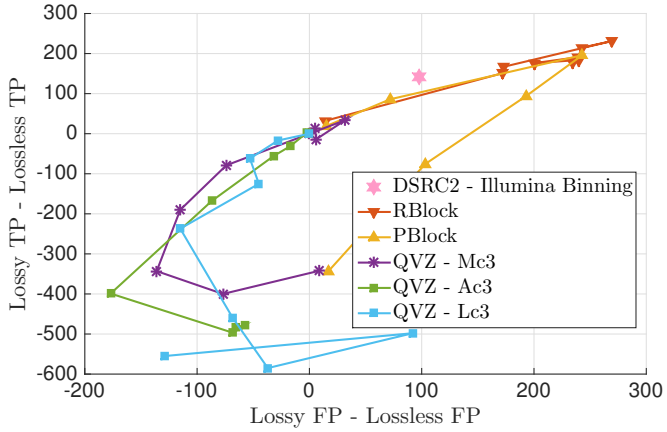


(c) Platypus pipeline

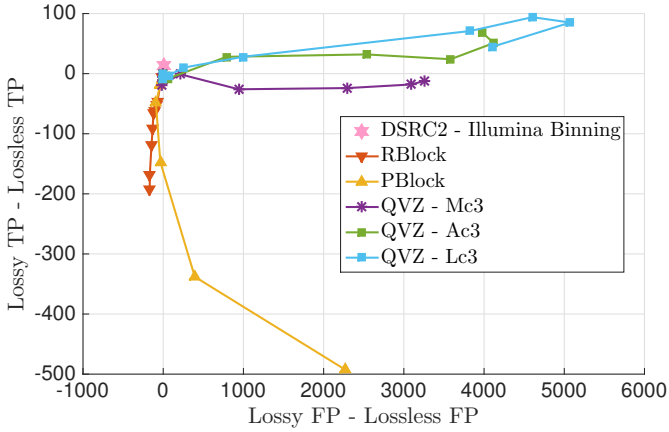
Figure 4: F.P. vs T.P. for different lossy compressors when using Illumina ground truth and the chromosome 11 of the NA12878 15× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.



(a) GATK pipeline



(b) *htslib.org* pipeline



(c) Platypus pipeline

Figure 5: F.P. vs T.P. for different lossy compressors when using Illumina ground truth and the chromosome 20 of the NA12878 15× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.

respect to the uncompressed after a certain rate, whereas in Platypus it achieves more F.P. and T.P. The number of T.P. remains similar as the rate increases, whereas the F.P. decrease.

Chromosome 20, NIST ground truth:

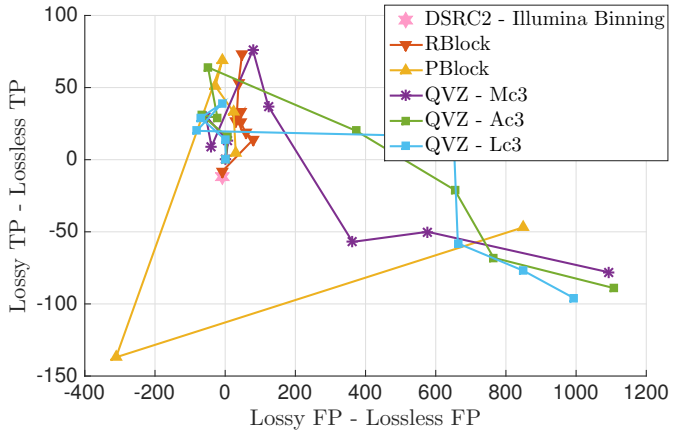
Fig. 7 shows the results of chromosome 20 of the 30× coverage dataset with the NIST ground truth. As can be observed, Illumina proposed binning performs similar to the chromosome 11, and it is again outperformed by other lossy compressors. Most of the RBlock points in GATK improve upon the uncompressed. The behavior in *htslib.org* and Platypus is similar to that of chromosome 11. PBlock in GATK also has several points that improve upon the uncompressed. In *htslib.org* and Platypus the behavior is very similar to chromosome 11. QVZ also offers a similar behavior to that observed in chromosome 11. The main difference (except for the scale of the axis, due to the smaller size of chromosome 20), is that in GATK QVZ gets more points that improve upon the uncompressed.

Chromosome 11, Illumina ground truth:

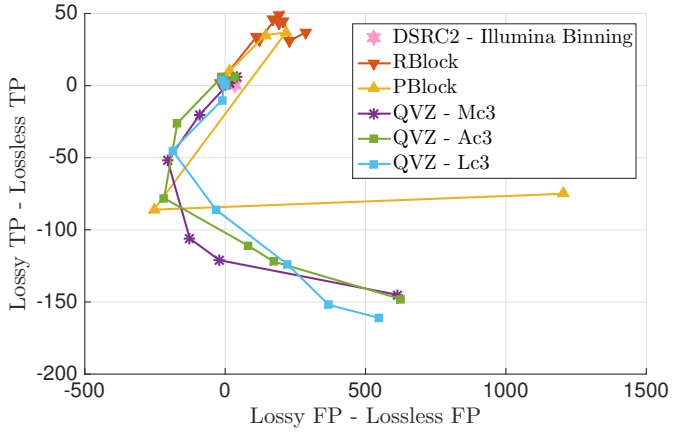
The results of using the Illumina ground truth are presented in Fig. 8. With the Illumina ground truth instead of NIST, in GATK most of the points move to the left and up, that is, they obtain fewer F.P. and more T.P. In *htslib.org* RBlock gets a better performance than with the NIST, PBlock as well, but only with the higher rates. Finally, QVZ does not seem to improve with respect to the NIST ground truth. On the contrary, PBlock and RBlock do not improve in Platypus, whereas QVZ does.

Chromosome 20, Illumina ground truth:

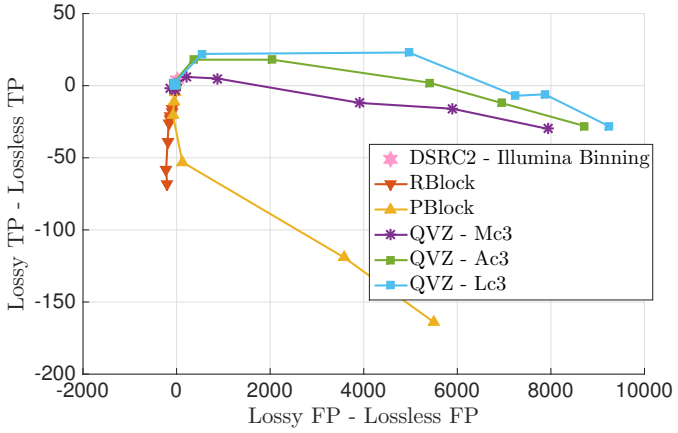
Fig. 9 shows the results of chromosome 20 with Illumina ground truth. With respect to the NIST ground truth, in GATK the lossy compressors offer a similar behavior, with fewer F.P. and more T.P. in general. In *htslib.org* Illumina has a very similar performance, RBlock improves,



(a) GATK pipeline

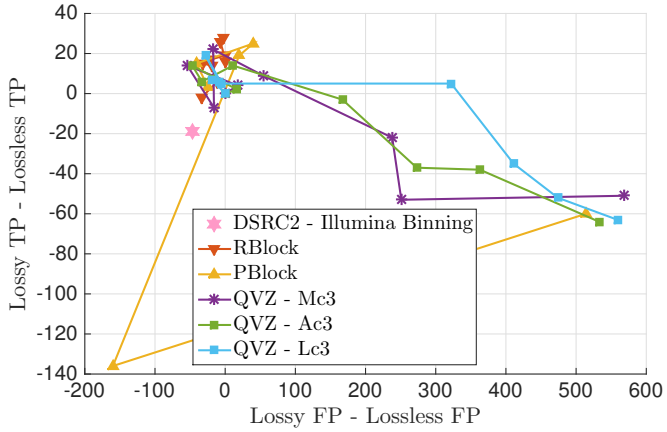


(b) *htslib.org* pipeline

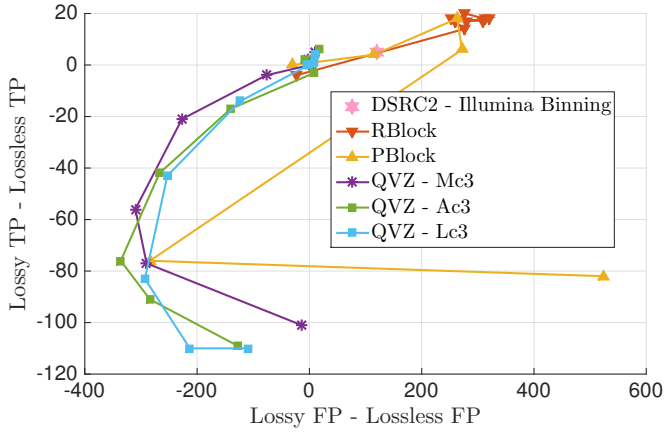


(c) Platypus pipeline

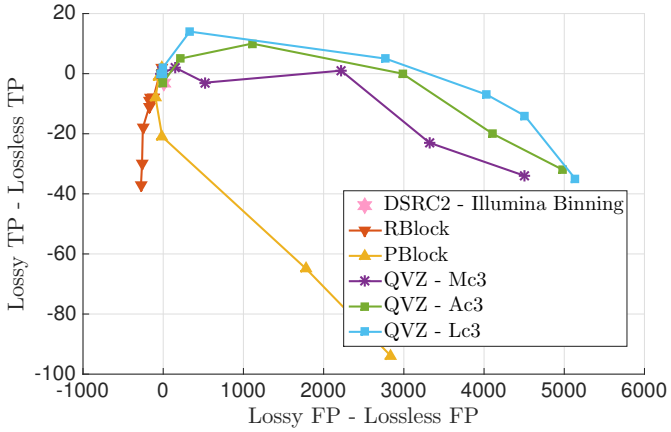
Figure 6: F.P. vs T.P. for different lossy compressors when using NIST ground truth and the chromosome 11 of the NA12878 30× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.



(a) GATK pipeline

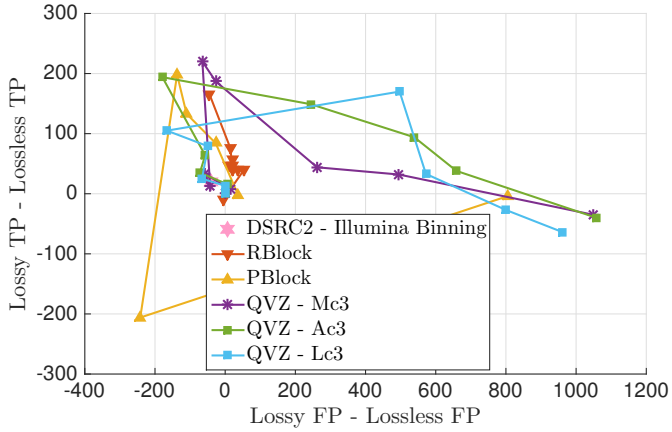


(b) *htslib.org* pipeline

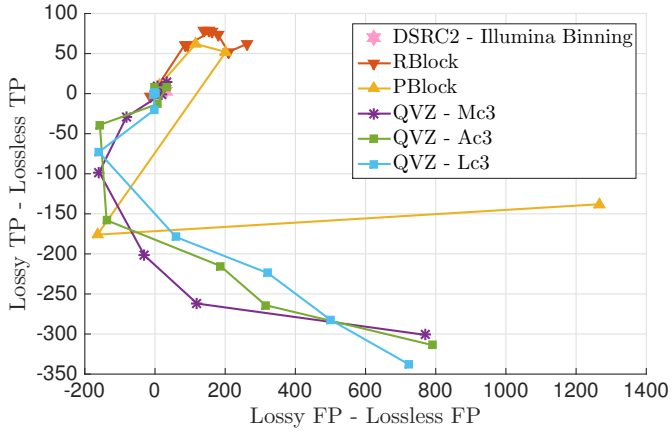


(c) Platypus pipeline

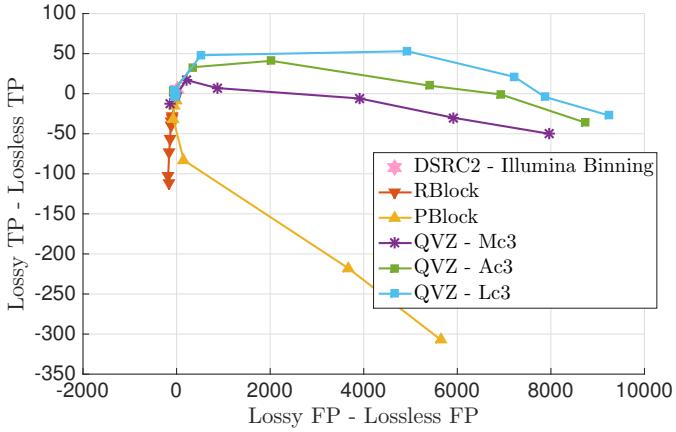
Figure 7: F.P. vs T.P. for different lossy compressors when using NIST ground truth and the chromosome 20 of the NA12878 30× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.



(a) GATK pipeline



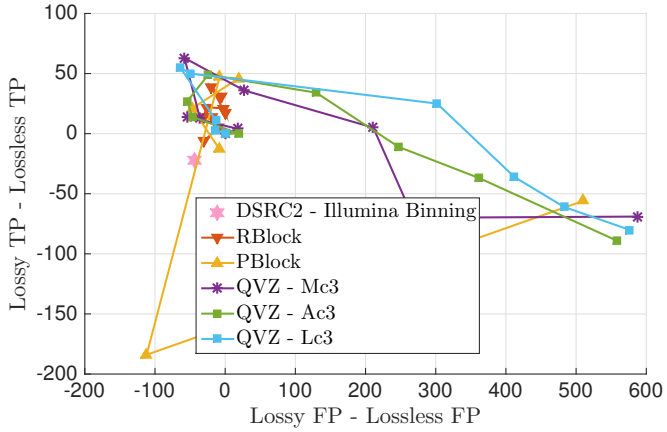
(b) *htslib.org* pipeline



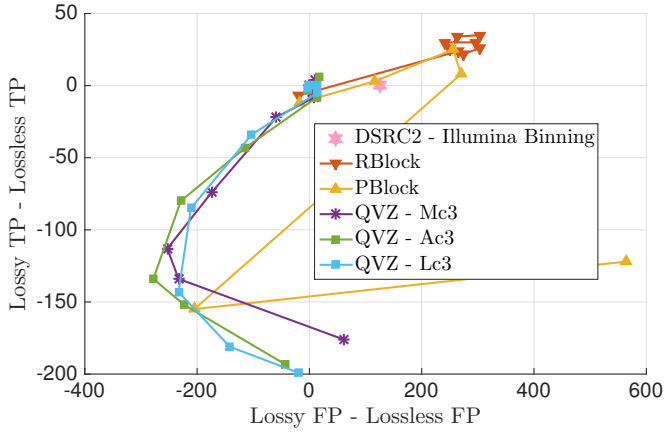
(c) Platypus pipeline

Figure 8: F.P. vs T.P. for different lossy compressors when using Illumina ground truth and the chromosome 11 of the NA12878 30× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.

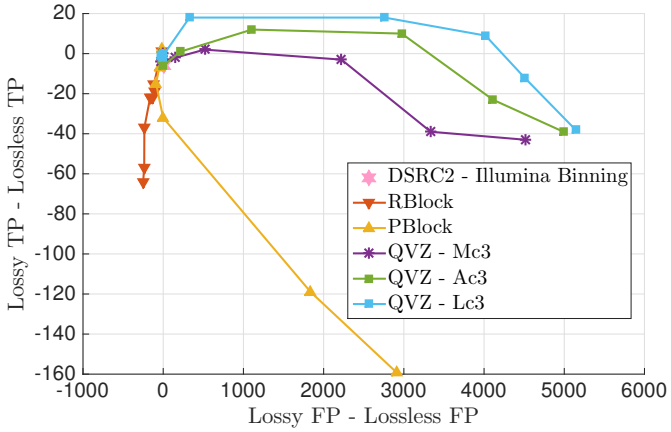
PBlock is slightly worse in small rates and almost identical at higher rates, and QVZ offers a slightly worse behavior. Finally, in Platypus Illumina improves, RBlock and PBlock deteriorate their performance, and QVZ improves.



(a) GATK pipeline



(b) *htslib.org* pipeline



(c) Platypus pipeline

Figure 9: F.P. vs T.P. for different lossy compressors when using Illumina ground truth and the chromosome 20 of the NA12878 30× coverage dataset, for the three considered pipelines. QVZ is shown with 3 clusters.

4.2 Performance in terms of sensitivity, precision and f-score

We provide tables that show the sensitivity, precision and f-score of the different lossy compressors for the NIST and Illumina ground truth, and for each of the considered pipelines. Results in red indicate an improvement with respect to the ones obtained with the original quality scores (uncompressed). The tables can be found in the *Supplementary Data (.xlsx)*. Specifically, *results_NIST_ground_truth* and *results_Illumina_ground_truth* contain the results of the NIST and Illumina ground truth, respectively. Each sheet shows the average results for the 30x coverage dataset (chromosomes 11 and 20), the 15x coverage dataset (chromosomes 11 and 20), and each of the individual chromosomes.

In the main paper we provided the discussion regarding the NIST ground truth, and thus here we focus on the results when using the Illumina ground truth. We observe that with the Illumina ground truth, the values of the sensitivity are smaller, whereas those of the precision are larger. These results can be easily explained by looking at the difference between the two ground truths (see Fig. 1. of the main paper). Illumina ground truth contains more SNPs than the NIST, and it also includes most of the SNPs contained on the NIST ground truth. Thus some of the calls that were F.P. with the NIST ground truth become T.P. with the Illumina ground truth, increasing the value of the precision. On the other hand, since the Illumina ground truth contains more SNPs, the sensitivity decreases.

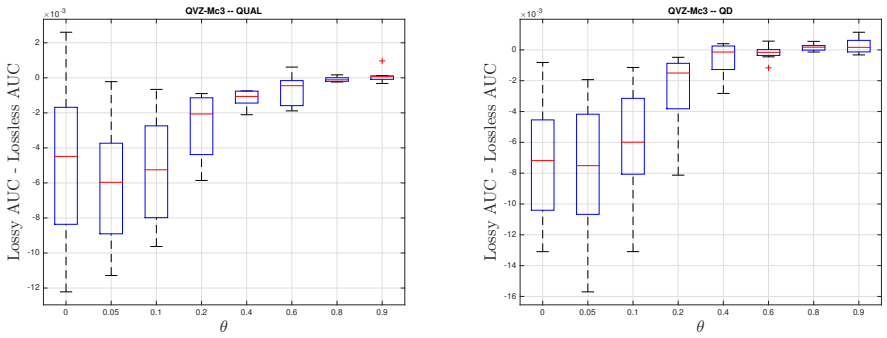
Regarding the GATK pipeline, we observe that for the 15x dataset the results are very similar (in terms of improving with respect to the uncompressed). For the 30× dataset we observe more differences. For example, in this case RBlock still improves in the sensitivity and f-score, but not in the precision. For the remaining algorithms the results are very similar to those obtain with the NIST ground truth. For the *htslib.org* pipeline, on the contrary, the results for the 30× coverage dataset are very similar, whereas more differences are observed in the 15× coverage dataset. For example, in

the latter case, RBlock and PBlock improve the f-score. Illumina proposed binning also improves in the f-score. Finally, the Platypus pipeline offers a very similar performance with the two ground truths, except for RBlock, that does not improve in general the f-score with the Illumina ground truth.

4.3 ROC performance

The ROC curve plots the False Positive Rate (F.P.R.) versus the True Positive Rate (T.P.R.) for different thresholding values. Specifically, given a thresholding parameter, only those SNPs in the VCF file whose parameter is above that threshold are considered positive calls. The remaining ones are considered negatives calls. By comparing the positive calls with the ground truth we compute the number of True Positives (T.P.) and False Positives (F.P.). Similarly, comparing the negative calls with the ground truth gives us the number of False Negatives (F.N.) and True Negatives (T.N.). Then, the F.P.R. is computed as F.P./(F.P. + T.N.), and the T.P.R. as T.P./(T.P. + F.N.). Varying the value of the threshold yields the ROC curve.

We encountered several problems when trying to compute the ROC curves. We first start with the selection of the thresholding parameter. To illustrate the problem, we compute the ROC curves for the QVZ algorithm with 3 clusters and MSE distortion criteria, for different rates, selecting as the thresholding parameters the QUAL and the QD fields of the VCF file. The results are shown in Fig. 10. Specifically, we show the box plot of the Area Under the Curve (AUC) differences between the lossy case and the lossless case. As it can be observed from the figure, the results are clearly different for the two thresholding parameters. Moreover, for each of the thresholding parameters different conclusions can be inferred. For example, take the rate $\theta = 0.8$. When the QUAL field is selected as the thresholding parameter, we see that the median is below zero, whereas with the QD it is above zero. On the contrary, for $\theta = 0$, better results are obtained with the QUAL parameter. This corroborates the fact that the selection of the thresholding parameter is important, as different results can be inferred for different parameters.



(a) Thresholding parameter: QUAL (b) Thresholding parameter: QD

Figure 10: Box plot of the difference between the lossy AUC and the lossless AUC, for the QVZ algorithm with 3 clusters and MSE distortion criteria, for different rates.

Another problem that we encountered is related to the fact that different VCF files contain a different number of calls. The approach followed in [4] was to analyze two VCF files at a time, by computing the union of the two, and adding the missing calls to each VCF file so that after their addition the two VCF files contained the same number of calls. The value of the selected parameter is set to zero for those calls that were not initially in a given VCF file. In our case, we would like to compare all the different lossy compressors, together with the different parameters (rate, cluster, etc.). Thus, instead of doing a pairwise comparison, we compare several VCF files following a similar approach as the one just described. The only difference is that we compute the union of all the VCF files under consideration. Then, as done in [4], we add the missing calls, with parameter set to zero, to each of the VCF files, so that at the end of this process all of them contain exactly the same calls. That is, the only difference between the set of calls (the completed VCF files) is the value of the thresholding parameter associated to each of the calls.

The approach described above to compare several VCF files using a ROC

Table 1: AUC for the lossless case (30× coverage dataset, chromosome 11), computing in different ways. Each column was computed by considering all the VCF files generated by QVZ with the number of clusters and distortion specify in the name of the column. For example, column MSE, c3 was computed by considering all the VCF files generated by QVZ with 3 clusters and MSE distortion.

Pipeline	Ground Truth	MSE, c3	L1, c3	MSE, c1
GATK	NIST	0.6053	0.6055	0.6113
GATK	Illumina	0.8229	0.8234	0.8339
htslib.org	NIST	0.6176	0.6180	0.6444
htslib.org	Illumina	0.8073	0.8081	0.8547
Platypus	NIST	0.6382	0.6492	0.6796
Platypus	Illumina	0.8544	0.8675	0.8963

curve may seem as the right approach. However, we noticed that depending of the VCF files that you compare at a time, different values of AUC are obtained. As an example, we computed the ROC curves considering the VCF files obtained with QVZ for a specific number of clusters and distortion metric, and different rates, including rate 1 (lossless compression). Specifically, we computed the ROC curves considering MSE and 3 clusters, L1 and 3 clusters, and MSE and 1 cluster. The results obtained for the lossless case (rate 1), which are the same in all cases, are summarized in Table 1. As it can be observed, the values of the AUC for the lossless cases when computing with the MSE and L1 with three clusters are different. This difference is more pronounce when comparing with the case of MSE with 1 cluster. For example, the AUC of the Platypus pipeline with the Illumina ground truth varies from 0.85 (MSE with 3 clusters) to 0.89 (MSE with 1 cluster).

This behavior makes the comparison of different AUCs unfair, unless all the VCF files that want to be compared are used to compute the union. The challenge here is that if we generate a new VCF file and want to compare

it with the previous generated VCF files, the whole analysis needs to be carried again. For these reasons, we believe it is better to compare VCF files by assuming all the calls are positive, and computing the sensitivity, precision and f-score, as done in the main manuscript.

However, for completeness, we decided to show here the results we obtained. Specifically, for each pipeline (GATK with hard filtering, *htslib.org* and Platypus), dataset and ground truth, we compute the ROC curve by considering all the VCF files generated by the different lossy compressors working at different conditions (rate, number of clusters, etc.). For that, we used the QUAL parameter as the thresholding parameter. Once the ROC curve is generated, we compute the AUC obtained with each of the VCF files under consideration.

In order to summarize the results, we computed for each case the difference of AUC with respect to the lossless case, such that a positive value is an improvement with respect to the uncompressed. Then, for each lossy compressor (working with specific parameters), we computed the average of this difference for the different pipelines and datasets. We did this computation for the two ground truths. The final results for the NIST and Illumina ground truths are shown in Fig 11 and Fig. 12, respectively. The x-axis represents the different lossy compressors, as specified by the legend. Given a lossy compressor, the points are sorted from smaller to higher rate. In the figure we show the box plot, as well as the average behavior.

With the NIST ground truth, we see that for a given lossy compressor, increasing the rate gives better AUCs in general. The variance also decreases with the rate, specially with QVZ. Regarding the QVZ algorithm, MSE distortion seems to work the best. PBlock achieves an average above zero for several rates, but it is generally outperformed by RBlock, which has an average above zero for all rates. For RBlock the median is above zero in some cases as well. Finally, Illumina's proposed binning has a large variance, with the median below zero and the average above.

With the Illumina ground truth we observe similar results. QVZ improves

its performance with the rate, and among the simulated distortions MSE seems to work the best. PBlock has again several points with average and median above zero, and RBlock offers the best performance. Illumina's point is similar to the previous case, but with the median above zero.

Finally, we computed the ROC curves when using the GATK with the VQSR filter pipeline. This pipeline only worked with the 30× coverage dataset (ERR262996), and thus we only show the results for this data for both chromosomes 11 and 20. The plots for chromosomes 11 and 20 and the NIST ground truth are shown in Fig. 13 and Fig. 14, respectively. Similarly, the results for the Illumina ground truth are shown in Fig. 15 (chromosome 11) and Fig. 16 (chromosome 20). In these plots we show the ROC performance of all the analyzed algorithms for all the rates and distortion in addition to the lossless performance, which is highlighted as a dashed red line. The aim of these plots is not to analyze individual performances but to have an overview of how the different lossy compressors affect the performance of the VQSR algorithm as a classifier.

As it can be observed in all the plots, for different points of the ROC curve different lossy compressors obtain the most true positive rate for the same false positive rate. That is, there is not a unique lossy compressor that works the best at each point. Interestingly, we also observe that the lossless curve is outperformed almost at all points by one of the lossy compressors. This observation is in line with the previous results, that showed that lossy compression can improve upon the uncompressed (or lossless compressed). Finally, one can also observe the effect of adding the missing calls to the VCF files with parameter value set to zero. This is specially pronounce for the chromosome 20 dataset, both for the NIST and Illumina ground truths, where one can see that the point previous to the (1,1) has a high true positive rate, but less than 0.5 of false positive rate. That is, most of the missing calls are false positives.

Overall, we can conclude that lossy compression has the potential to offer a performance close or better to that obtain with the original quality scores,

while significantly reducing the size of the files.

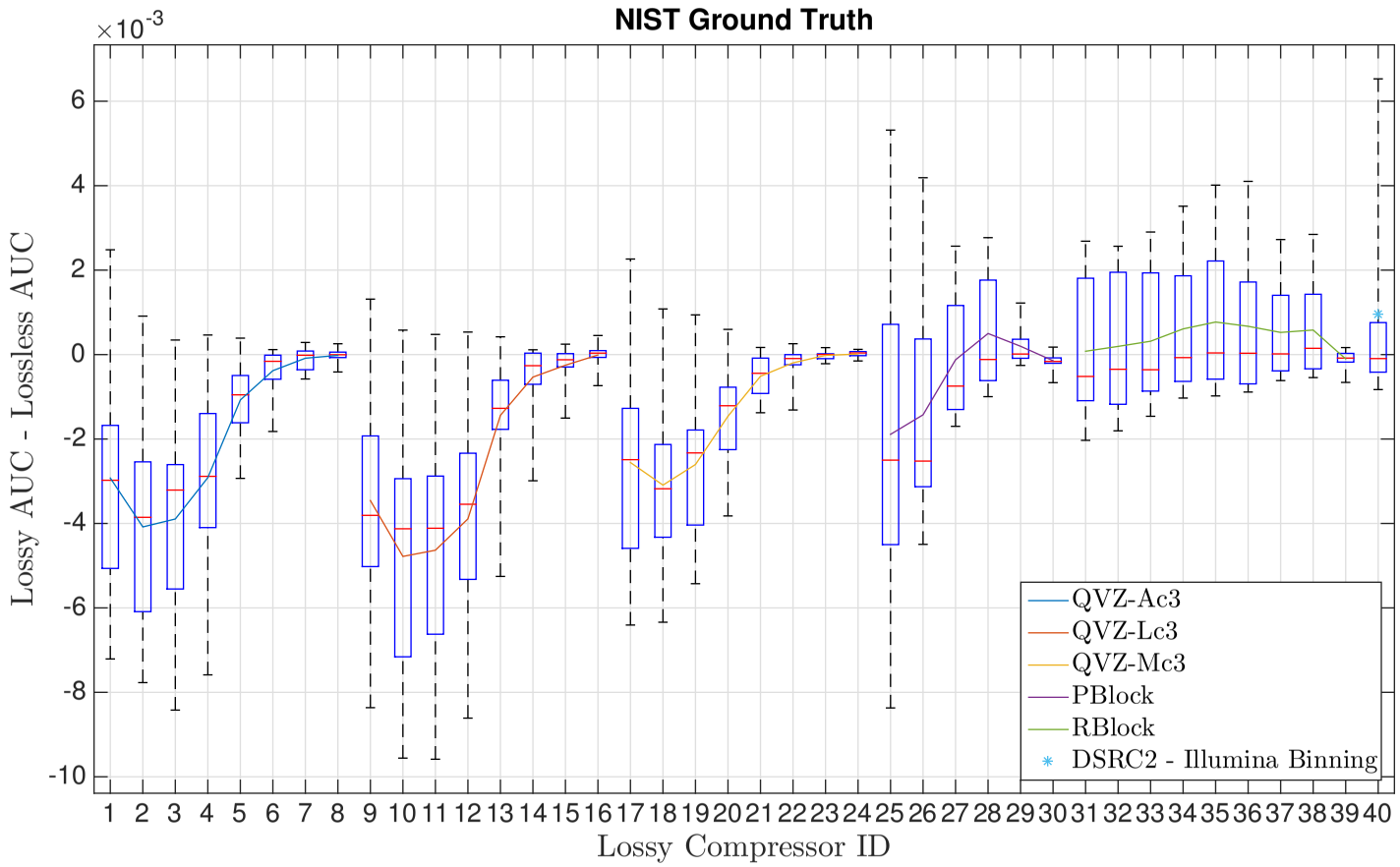


Figure 11: Box plot and average behavior of the different lossy compressors, in terms of AUC, when using the NIST ground truth. The results are normalized with those of the lossless case, such as a value above 0 represents an AUC bigger than that obtained with the lossless file. The points within a lossy compressor are sorted from smaller to higher rate.

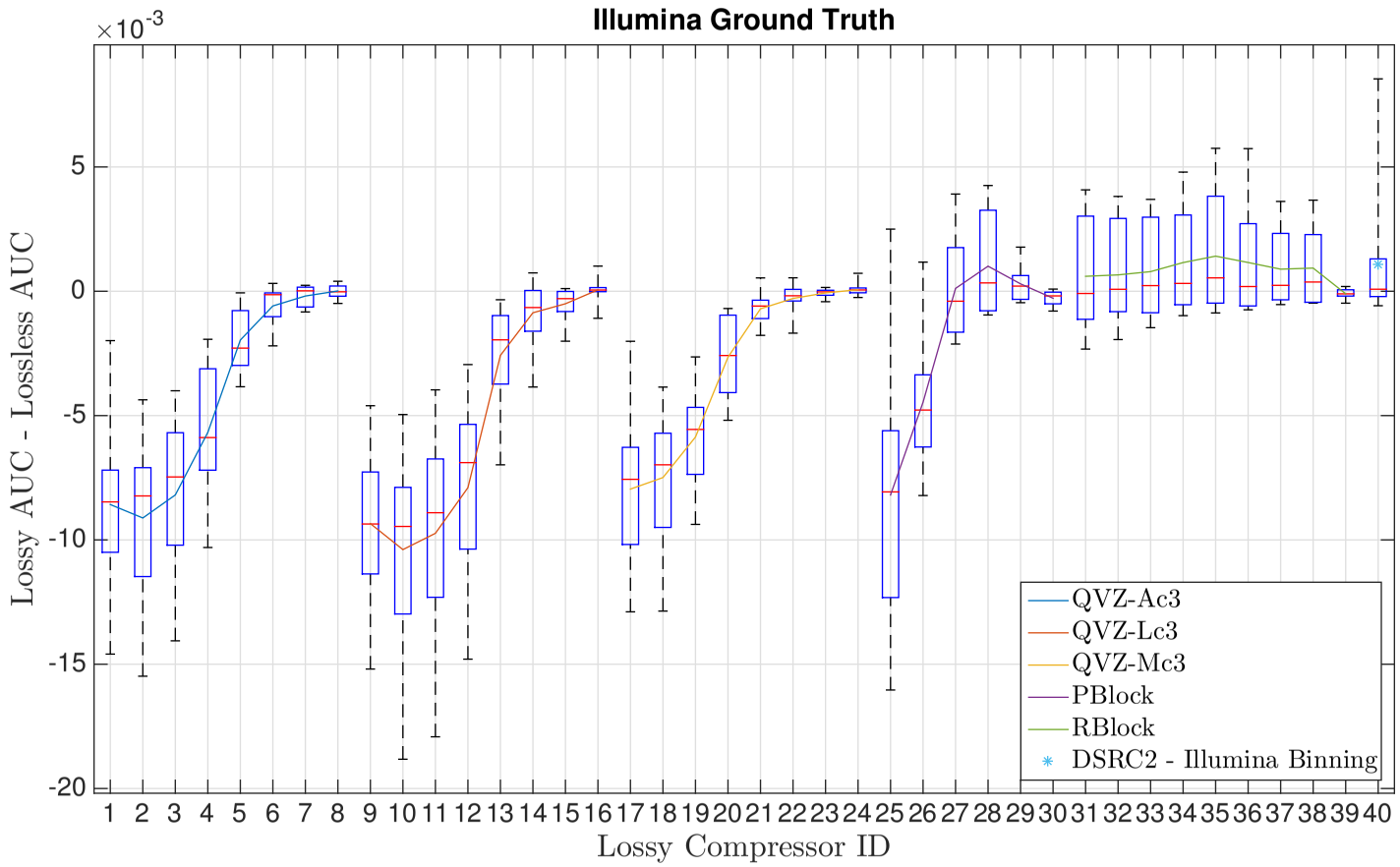


Figure 12: Box plot and average behavior of the different lossy compressors, in terms of AUC, when using the Illumina ground truth. The results are normalized with those of the lossless case, such as a value above 0 represents an AUC bigger than that obtained with the lossless file. The points within a lossy compressor are sorted from smaller to higher rate.

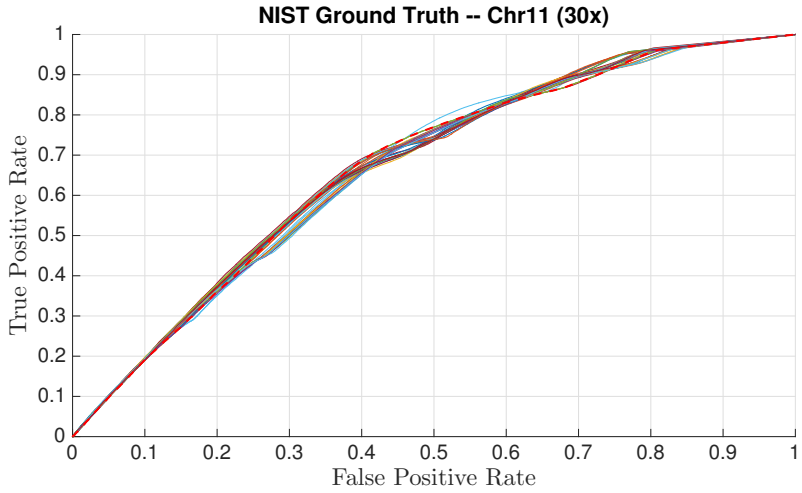


Figure 13: ROC plot for the GATK pipeline with VQSR filter and thresholding parameter VQSLOD, with the NIST ground truth and dataset chromosome 11 (30× coverage dataset).

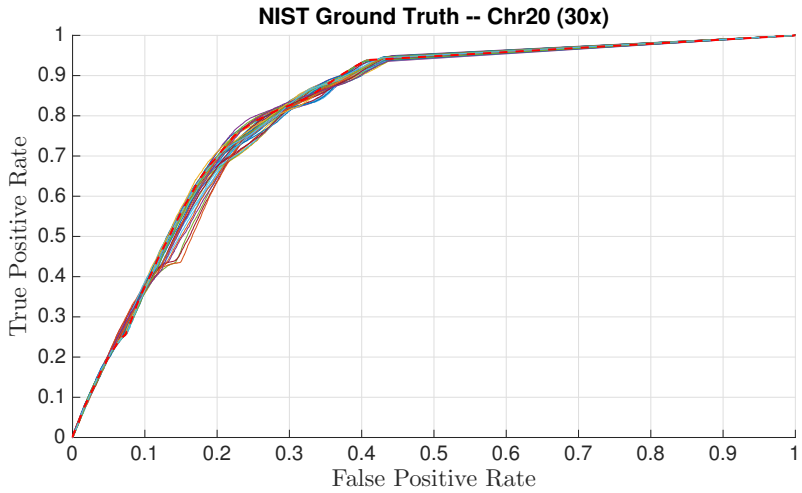


Figure 14: ROC plot for the GATK pipeline with VQSR filter and thresholding parameter VQSLOD, with the NIST ground truth and dataset chromosome 20 (30× coverage dataset).

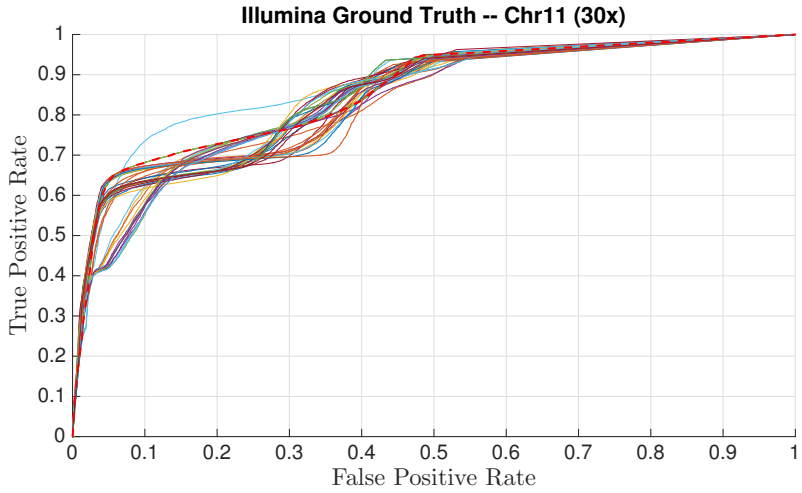


Figure 15: ROC plot for the GATK pipeline with VQSR filter and thresholding parameter VQSLOD, with the Illumina ground truth and dataset chromosome 11 (30× coverage dataset).

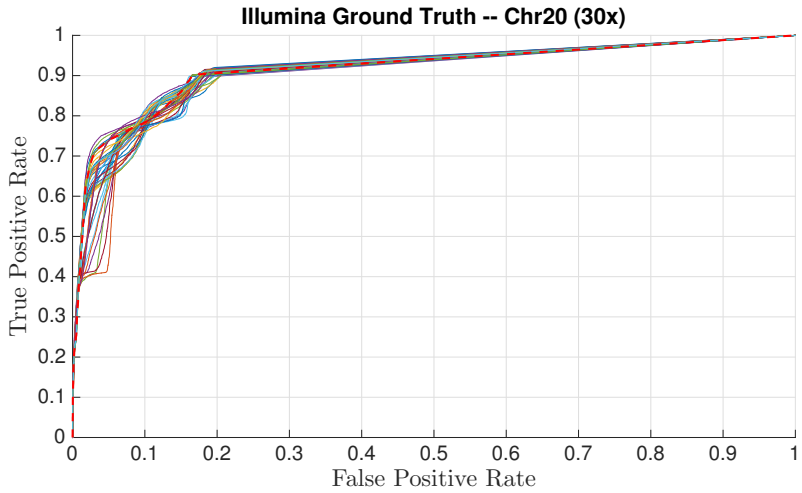


Figure 16: ROC plot for the GATK pipeline with VQSR filter and thresholding parameter VQSLOD, with the Illumina ground truth and dataset chromosome 20 (30× coverage dataset).

References

[1] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly *et al.*, "The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[2] R. Cánovas, A. Moffat, and A. Turpin, "Lossy compression of quality scores in genomic data," *Bioinformatics*, vol. 30, no. 15, pp. 2130–2136, 2014.

[3] G. Malysa, M. Hernaez, I. Ochoa, M. Rao, K. Ganesan, and T. Weissman, "Qvz: lossy compression of quality values," *Bioinformatics*, p. btv330, 2015.

[4] Y. W. Yu, D. Yorukoglu, J. Peng, and B. Berger, "Quality score compression improves genotyping accuracy," *Nature biotechnology*, vol. 33, no. 3, pp. 240–243, 2015.