GENETIC VARIANT CLASSIFICATION

Project report submitted to
Guru Jambheshwar University of Science and Technology, Hisar
for the partial award of the degree

of

Bachelor of Technology
in
Computer Science and Engineering

by

Arvind (200010130018)
Ashish Kumar (200010130020)

Dr. Yogesh Chaba
Professor

Department of Computer Science & Engineering
GURU JAMBHESHWAR UNIVERSITY OF SCIENCE AND TECHNOLOGY, HISAR
June, 2024

DECLARATION

We, Arvind (200010130018) and Ashish Kumar (200010130020), certify that the work contained in this project report is original and has been carried us under the guidance of my supervisor. This work has not been submitted to any other institute for the award of diploma and I have followed the ethical practices and other guidelines provided by the Department of Computer Science and Engineering in preparing the report. Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the report and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Arvind
200010130018
Department of Computer Science and Engineering
Guru Jambheshwar University of Science and Technology, Hisar

Ashish Kumar
200010130020
Department of Computer Science and Engineering
Guru Jambheshwar University of Science and Technology, Hisar

Dr. Yogesh Chaba
Professor
Department of Computer Science and Engineering
Guru Jambheshwar University of Science and Technology, Hisar

ACKNOWLEDGEMENT

This is to acknowledge that our project would not have been possible without the support and help of individuals who helped me in making this a successful project. I would like to extend my sincere thanks to all of them.

I am highly grateful and indebted to **Dr. Yogesh Chaba** for his supervision and guidance for providing important and necessary information related to the project time to time and for his support in completion of this project.

I would like to express my gratitude towards my parents for promoting me to this level and help me in completion of the project.

Also, I would like to thank my colleagues who directly or indirectly helped me in the development of this project and provide necessary feedback from time to time.

Arvind (200010130018) &

Ashish Kumar (200010130020)

CERTIFICATE

This is to certify that Arvind having roll no. 200010130018 and Ashish Kumar having roll no. 200010130020 are students of B.Tech (CSE-I), Department of Computer Science and Engineering, Guru Jambheshwar University of Science and Technology, Hisar have completed the project entitled “Genetic Variant Classification”.

Dr. Yogesh Chaba
Professor
Department of Computer Science and Engineering
Guru Jambheshwar University of Science and Technology, Hisar

CONTENT

ABSTRACT

This study aimed to elucidate the relationship between the pathogenicity of genetic mutations and the classification of the resulting diseases, utilizing the ClinVar database. An intensive data cleaning process was performed, including handling missing values, detecting outliers, and eliminating irrelevant columns. During this phase, a critical observation was made: the allele frequencies for genetic mutations varied significantly across different databases. This discrepancy poses a significant challenge in the field of genomics, as it could potentially lead to misleading interpretations and outcomes in genetic disease classification.

Despite this obstacle, various machine learning models were deployed on the refined dataset. Initial models, however, struggled due to the imbalanced 'CLASS' target variable, achieving a less than ideal accuracy of 76% and an R-squared of -0.25.
Nevertheless, the exploration of the PolyPhen variable and 'Consequence' column yielded more promising results. The XGBoost model, when applied to the PolyPhen variable, reached an accuracy of 78.8% and an R-squared of 0.21. Further, the model targeted towards the 'Consequence' column resulted in an impressive accuracy of 93.4% and an R-squared of 0.957, indicating a strong model fit.

In conclusion, our study underscores the complexities involved in genetic mutation classifications, including significant variances in allele frequencies across databases. Despite early challenges, our research demonstrated the robust potential of machine learning in decoding these intricacies and provided a solid foundation for future research in genetic disease classification.

CHAPTER 1
INTRODUCTION

1.1 Introduction to Project

Genetic variant classification is a critical process in the field of genetics, particularly in a diagnostic setting. It forms the basis for clinical decision making and is crucial for managing patients and realizing the best possible outcomes.

The classification of genetic variants involves assigning categories to these variants based on their potential impact on health or disease. The process entails the collation and evaluation of various sources of evidence to determine the clinical significance of variants identified through diagnostic testing for a disease with a suspected underlying genetic cause.

Most genetic diagnostics laboratories develop and use their own in-house variant classification system, with many following the recommendations of the American College of Medical Genetics and Genomics (ACMG) guidelines. However, there can be differences between the classification systems, which can sometimes result in different classifications of the same variant between companies.

Blueprint Genetics, for example, has developed a variant classification system intended to classify variants in dominant monogenic disorders, which are rare diseases caused by single variants in single genes. Their system closely follows the guidelines and interpretation criteria established by the ACMG.

The classification system typically involves a 5-tiered scheme that describes the quantity and quality of evidence needed to classify a genetic variant as pathogenic, likely pathogenic, a variant of uncertain significance (VUS), likely benign, or benign. This systematic, clear, and sensible variant evaluation criteria is crucial for making confident diagnostic decisions.

1.2 Problem Formulation

The toxicity of the resulting genetic anomalies has a significant impact on the classification of ensuing disorders; many mutations result in benign ailments or have no discernible effects, while others create serious or potentially deadly issues. The classification of genes and the degree of severity and clinical manifestation of the disorders they correspond with are often based on the functional implications of mutations, such as nonsense mutations that produce truncated, non-functioning proteins or missense mutations that alter the structure of proteins. Understanding this connection is essential to approaches including targeted and tailored therapy.

1.3 Objectives

- **Functional Impact on Protein Structure:** Highlight the crucial role of genetic modifications in altering protein structure, emphasizing the distinction between benign mutations and those with serious implications. Clarify that missense mutations may lead to changes in protein structure, while nonsense mutations result in the production of non-functional proteins, influencing the severity of associated disorders.
- **Impact on Severity and Clinical Manifestation:** Emphasize the direct correlation between the toxicity of genetic anomalies and the classification of resulting disorders. Illustrate how the degree of severity and clinical manifestation of diseases is often determined by the nature of mutations, providing a spectrum from benign conditions to potentially life-threatening issues.
- **Importance for Targeted and Tailored Therapy**: Stress the significance of understanding the relationship between pathogenicity and disease classification for the development of targeted and tailored therapeutic interventions. Highlight that insights into the functional implications of genetic mutations enable more precise and effective approaches in personalized medicine, enhancing the potential for successful treatment strategies.

CHAPTER 2
BACKGROUND DETAILS AND LITERATURE REVIEW

2.1 Background Details

Genetic variation refers to the differences in DNA sequences among individuals, which can affect physical traits, susceptibility to diseases, and response to medications. These variations can range from single nucleotide polymorphisms (SNPs) to larger structural changes like insertions, deletions, and copy number variations. Understanding and classifying these variations is crucial for medical genetics, particularly in diagnosing and treating genetic disorders.

The pathogenicity of a genetic variant refers to its ability to cause disease. Variants can be classified as benign, likely benign, uncertain significance (VUS), likely pathogenic, or pathogenic. The classification depends on the variant's impact on gene function and its association with clinical symptoms. Pathogenic variants are often linked to genetic disorders, while benign variants typically have no adverse effects on health.

The Exome Sequencing Project (ESP) from the National Heart, Lung, and Blood Institute (NHLBI) aims to discover and characterize genetic variants in the human genome. The GO-ESP dataset provides valuable information on genetic variants from a large cohort of individuals, making it a critical resource for studying genetic variation and its implications for human health.

The Exome Aggregation Consortium (ExAC) dataset is one of the most comprehensive resources for understanding genetic variation in the human exome. It aggregates exome sequencing data from a diverse range of populations to provide a detailed catalog of human genetic variation. This dataset has been instrumental in the study of rare genetic variants and their implications for human health. Exome sequencing focuses on the protein-coding regions of the genome, which represent about 1-2% of the human genome but contain approximately 85% of known disease-related variants. This makes exome sequencing a powerful tool for identifying genetic variants associated with diseases.

The 1000 Genomes Project (TGP) was an international collaboration aimed at creating the most detailed catalog of human genetic variation. Launched in 2008, the project sequenced the genomes of over 2,500 individuals from various populations around the world. This comprehensive dataset provides a valuable resource for researchers studying human genetics, disease association, and population genomics. Unlike exome sequencing, which targets only the protein-coding regions of the genome, whole genome sequencing (WGS) captures the complete DNA sequence of an individual. This includes both coding and non-coding regions, providing a more comprehensive view of genetic variation.

Conflicting classifications are when two of any of the following three categories are present for one variant, two submissions of one category are not considered conflicting.

1. Likely Benign or Benign
2. VUS
3. Likely Pathogenic or Pathogenic

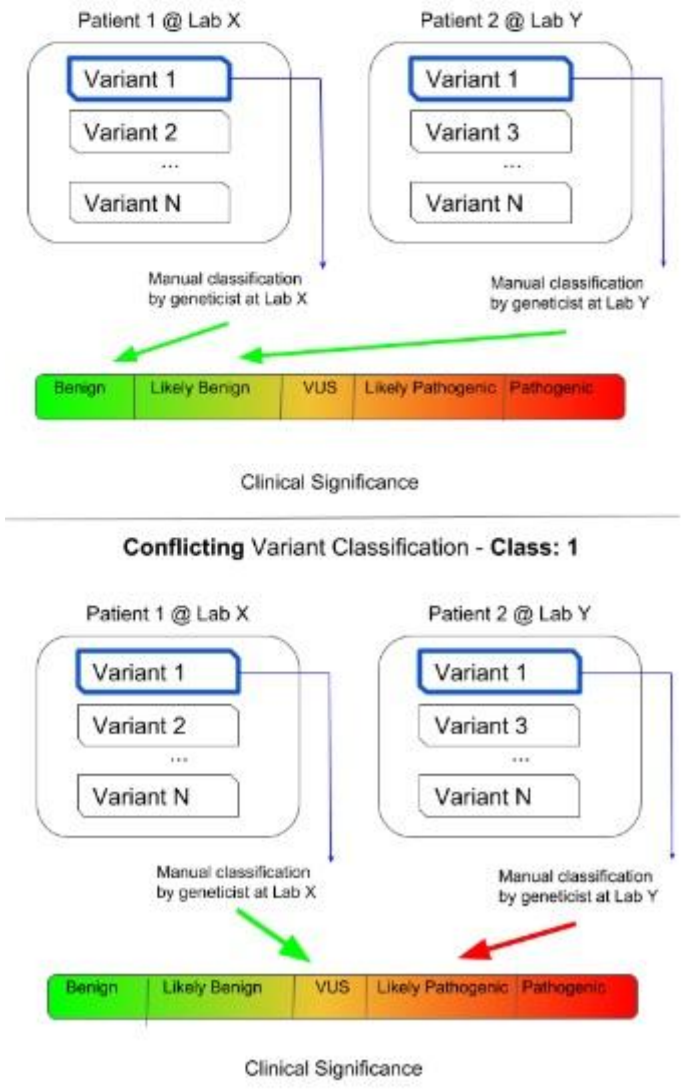


Fig. 2.1 Concordant and Conflicting Variants

2.2 Literature Review

Recent studies have focused on improving the methods for classifying genetic variants, leveraging large datasets and advanced computational techniques. A common approach involves the use of machine learning algorithms trained on labeled datasets to predict the pathogenicity of new variants. These models consider various features, such as evolutionary conservation, biochemical properties, and population frequency.

ClinVar has been instrumental in many research studies aimed at understanding the genetic basis of diseases. For example, Richards et al. (2015) provided guidelines for interpreting sequence variants, which have been widely adopted by clinical laboratories for consistent classification. The ClinVar dataset serves as a benchmark for evaluating new computational tools and methods developed for variant classification.

Advances in machine learning have led to the development of several tools for predicting the pathogenicity of genetic variants. Tools like PolyPhen-2, SIFT, and CADD use different algorithms and input features to assess the potential impact of variants. These tools are often validated using ClinVar data to ensure their accuracy and reliability.

Despite significant progress, challenges remain in the accurate classification of genetic variants. Variants of uncertain significance (VUS) represent a substantial portion of the variants in ClinVar, reflecting the limitations in current knowledge and data. Integrating multiple lines of evidence, such as functional assays, co-segregation studies, and computational predictions, is essential for improving variant interpretation.

Recent approaches emphasize the integration of genomic data with other types of information, such as phenotypic data, electronic health records, and population-specific data. Studies like those by Landrum et al. (2016) highlight the importance of combining diverse datasets to enhance the accuracy of pathogenicity predictions and facilitate personalized medicine.

Ongoing research aims to address the challenges associated with variants of uncertain significance and improve the integration of multidimensional data to enhance the interpretation of genetic variants. Understanding and leveraging ClinVar data will continue to be essential in advancing the field of medical genetics and improving patient outcomes.

CHAPTER 3
SOFTWARE REQUIREMENTS AND SPECIFICATIONS

3.1 Software Requirements

3.1.1 Operating System: Windows7/8/10/11

3.1.2 Programming Language: Python

3.1.3 Pycharm: Pycharm is an integrated development environment(IDE) designed specifically for Python programming. Developed by JetBrains, it provides comprehensive features to improve our Python development productivity.Here are some of the important features and capabilities of Pycharm:

- Code Editor
- Debug and Test
- Integrated version control
- Project Management
- Refactoring and Code Analysis
- Integrated Terminal
- Third party libraries and frameworks

3.1.4 Visual Studio Code

Visual Studio Code (VS Code) is a popular open-source code editor developed by Microsoft. It is widely used by developers across multiple programming languages and platforms. The editor is available for Windows, macOS, and Linux and provides support for a wide range of programming languages, including Java, Python, C++, and JavaScript.

One of the key features of VS Code is its intuitive user interface, which makes it easy to use for both experienced and beginner developers. The editor has a customizable layout, with support for multiple tabs, split views, and an integrated terminal. It also supports a wide range of extensions, which can be installed to add additional functionality to the editor.

VS Code includes a range of features that make it popular among developers, such as:

1. Intelligent code completion: VS Code provides intelligent code completion, which suggests code snippets based on the context of the code. This feature

makes coding faster and more efficient.

2. Debugging: VS Code includes a powerful debugger, which makes it easy to debug code and track down errors.

3. Git integration: VS Code integrates seamlessly with Git, which makes it easy to manage source code repositories and collaborate with other developers.

4. Language support: VS Code supports a wide range of programming languages, and includes features like syntax highlighting, code folding, and autocompletion for each language.

5. Extensions: VS Code includes a wide range of extensions, which can be installed to add additional functionality to the editor. There are extensions available for almost every programming language and use case.

In addition to these features, VS Code also has a vibrant community of developers who contribute to its development and create new extensions. The editor is open-source and available on GitHub, which makes it easy for developers to contribute to the project.

In summary, VS Code is a powerful and versatile code editor that is popular among developers across multiple platforms and programming languages. Its intuitive user interface, intelligent code completion, debugging features, Git integration, and wide range of extensions make it a great choice for any developer looking for a reliable and efficient code editor.

3.1.5 Jupyter

Jupyter is an open-source web-based environment that allows for the creation and sharing of interactive documents containing live code, equations, visualizations, and narrative text. Jupyter notebooks are widely used in data science and scientific research communities due to their versatility, interactivity, and ease of use.

The Jupyter environment is built around a notebook interface, which allows users to create and share documents that contain live code, narrative text, and visualizations.

The notebooks are organized into cells, which can contain code, text, or visualizations. The cells can be executed in any order, and the results of each cell are displayed directly below it.

One of the key benefits of the Jupyter environment is its support for multiple programming languages, including Python, R, Julia, and Scala. This allows users to work with a variety of data analysis and scientific computing tools, and to seamlessly integrate code from different languages in a single notebook.

Another advantage of the Jupyter environment is its ability to produce interactive visualizations, which allow users to explore and manipulate data in real-time. The environment includes support for a variety of visualization libraries, including matplotlib, ggplot, and bokeh.

Jupyter also supports the creation and sharing of notebooks with others. Notebooks can be shared online or saved as files, which can be easily emailed or uploaded to a website. This makes it easy for teams to collaborate on data analysis projects, and for researchers to share their work with others.

In summary, the Jupyter environment is a powerful and versatile tool for data analysis and scientific computing. Its support for multiple programming languages, interactive visualizations, and collaborative features make it a popular choice among data scientists and researchers.

3.2 Hardware Requirements

The following are the hardware required for the project:

- PC with Pentium II Processor,450MHz (Recommended Pentium III Processor,800MHz)
- 4GB RAM
- Minimum 1.2 GB magnetic disk space.
- PC should be connected with Network.
- CD-ROM (48 X or higher recommended).
- Mouse or Similar Pointing device.
- A Printer is required to take out Reports.

CHAPTER 4
DESIGN OF THE PROJECT REPORT

4.1 Methodology

Genetic Variant Classification of conflicting variants use the following methodologies:

4.1.1 Dataset

The ClinVar dataset, managed by the National Center for Biotechnology Information (NCBI), is a crucial archive that links human genetic variations to associated phenotypes. It is an invaluable resource for researchers, healthcare professionals, and clinicians interpreting genetic variants in the context of health and disease. The dataset include the following features:

- CHROM – Chromosome name or number on which the variant is located
- POS – Position on the chromosome the variant is located on.
- REF – Reference Allele
- ALT – Alternate Allele
- AF_ESP - Allele frequencies from GO-ESP
- AF_EXAC - Allele frequencies from ExAC
- AF_TSP - Allele frequencies from the 1000 genomes project
- CLNDISDB - Tag-value pairs of disease database name and identifier
- CLNDISDBINCL - For included Variant: Tag-value pairs of disease database name and identifier
- CLNDN - ClinVar's preferred disease name for the concept specified by disease identifiers in CLNDISDB
- CLNDNINCL - For included Variant : ClinVar's preferred disease name for the concept specified by disease identifiers in CLNDISDB
- CLNHGVS - Top-level (primary assembly, alt, or patch) HGVS expression
- CLNSIGINCL - Clinical significance for a haplotype or genotype that includes this variant. Reported as pairs of VariationID:clinical significance.
- CLNVC – Variant Type
- CLNVI - the variant's clinical sources reported as tag-value pairs of database and variant identifier
- MC - comma separated list of molecular consequence in the form of Sequence Ontology ID|molecular_consequence
- ORIGIN - Allele origin. One or more of the following values may be added: 0 - unknown; 1 - germline; 2 - somatic; 4 - inherited; 8 - paternal; 16 - maternal; 32 - de-novo; 64 - biparental; 128 - uniparental; 256 - not-tested; 512 - tested-inconclusive; 1073741824 – other

- SSR - Variant Suspect Reason Codes. One or more of the following values may be added: 0 - unspecified, 1 - Paralog, 2 - byEST, 4 - oldAlign, 8 - Para_EST, 16 - 1kg_failed, 1024 – other
- CLASS - The binary representation of the target class. 0 represents no conflicting submissions and 1 represents conflicting submissions.
- Allele - the variant allele used to calculate the consequence
- Consequence - Type of consequence: https://useast.ensembl.org/info/genome/variation/prediction/predicted_data.html#consequences
- IMPACT - the impact modifier for the consequence type
- SYMBOL – Gene Name
- Feature_type - type of feature. Currently one of Transcript, RegulatoryFeature, MotifFeature.
- Feature - Ensembl stable ID of feature
- BIOTYPE - Biotype of transcript or regulatory feature
- EXON - the exon number (out of total number)
- INTRON - the intron number (out of total number)
- cDNA_position - relative position of base pair in cDNA sequence
- CDS_position - relative position of base pair in coding sequence
- Protein_position - relative position of amino acid in protein
- Amino_acids - only given if the variant affects the protein-coding sequence
- Codons - the alternative codons with the variant base in upper case
- DISTANCE - Shortest distance from variant to transcript
- STRAND - defined as + (forward) or - (reverse)
- BAM_EDIT - Indicates success or failure of edit using BAM file
- SIFT - the SIFT prediction and/or score, with both given as prediction(score)
- PolyPhen - the PolyPhen prediction and/or score
- MOTIF_NAME - the source and identifier of a transcription factor binding profile aligned at this position
- MOTIF_POS - The relative position of the variation in the aligned TFBP
- HIGH_INF_POS - a flag indicating if the variant falls in a high information position of a transcription factor binding profile (TFBP)
- MOTIF_SCORE_CHANGE - The difference in motif score of the reference and variant sequences for the TFBP
- LoFtool - Loss of Function tolerance score for loss of function variants: https://github.com/konradjk/loftee

- CADD_PHRED - Phred-scaled CADD score
- CADD_RAW - Score of the deleteriousness of variants: http://cadd.gs.washington.edu/
- BLOSUM62 - See: http://rosalind.info/glossary/blosum62/

ClinVar aggregates data from clinical testing laboratories, research institutions, and expert panels. The dataset includes details about genetic variants such as single nucleotide polymorphisms (SNPs), insertions, deletions, and complex rearrangements. Each variant is described by its genomic coordinates, reference sequence, and nucleotide or amino acid changes. Clinical significance annotations classify variants as "benign," "likely benign," "uncertain significance," "likely pathogenic," or "pathogenic." The dataset also details associations between variants and specific health conditions, including disease names, clinical features, and inheritance patterns. Supporting evidence from publications, clinical reports, and experimental studies, along with submitter information, ensures transparency and data credibility.

ClinVar is a diagnostic tool for clinicians, aiding in the identification of genetic conditions by comparing patient data with known variants. Researchers use it to explore genotype-phenotype correlations, identify therapeutic targets, and understand disease genetics. Additionally, it serves as an educational resource for genetic counselors, medical students, and healthcare professionals.

Challenges for ClinVar include ensuring consistent annotations, resolving interpretation discrepancies, and integrating emerging genomic data. Future improvements aim to enhance accuracy, expand content, and develop advanced tools for data analysis.

ClinVar is an indispensable resource in medical genetics, providing a comprehensive repository of genetic variants and their clinical implications. Its ongoing development is vital for advancing personalized medicine and deepening our understanding of the genetic basis of diseases.

4.1.2 Data Cleaning:

In order to ensure the integrity and reliability of subsequent analyses, it is imperative to conduct meticulous data cleaning procedures. Data cleaning entails comprehensive examination of null values, identification of outliers, and initial visualization of relationships between categorical and continuous variables. By undertaking these essential steps, one can mitigate the potential for inaccuracies in feature selection and model generation, thereby establishing a solid foundation for downstream analyses and insights. Features having high percentage of missing values with 99% missing threshold are excluded. A heatmap show the relationship between different numerical features. Unnecessary and redundant features are eliminated. Following are the unnecessary variables:

- BAM_EDIT
- INTRON
- EXON
- CLNDISDB
- CLNHGVS
- MC
- CLNVI
- SYMBOL
- BIOTYPE
- CADD_RAW
- Allele
- Feature
- Feature_type

Features with a few missing values are forward filled and interpolated.

4.1.3 Data Exploratory Analysis:

EDA is an approach of analyzing data sets to summarize their main characteristics, often using statistical graphics and other data visualization methods. Regarding the 'REF' and 'ALT' columns, it is customary for these values to consist of a single letter representing one of the four nucleotide bases: G, C, A, T. Upon examination, it is evident that these columns have excessive number of unique values. This can be attributed to anomalous entries where multiple letters are present. Consequently, rows containing such multi-letter values will be excluded. Factorized encoding is used to assign a numerical value to every unique value in the object columns without introducing any sort of rank or compromising dimensionality.

4.1.4 Feature Engineering:

Feature engineering is the process of using domain knowledge to create new features or modify existing features in a dataset to improve the performance of machine learning models. It is a critical step in the data preprocessing pipeline and can significantly influence the effectiveness of the models. Here's a detailed look at feature engineering:

Why Feature Engineering Matters

- Improves Model Accuracy: Well-engineered features can make patterns in the data more apparent to machine learning algorithms, leading to better predictions.
- Reduces Model Complexity: By transforming and combining raw data, feature engineering can simplify the learning task, making models more efficient.

- Addresses Data Limitations: It helps in handling missing data, reducing dimensionality, and improving data quality.

4.1.5 Model Selection and Training:

Following algorithms and models are selected and trained on the preprocessed data.

1. Chi-Squared - A chi-squared test (also chi-square or χ^2 test) is a statistical hypothesis test used in the analysis of contingency tables when the sample sizes are large. In simpler terms, this test is primarily used to examine whether two categorical variables (two dimensions of the contingency table) are independent in influencing the test statistic (values within the table). The test is valid when the test statistic is chi-squared distributed under the null hypothesis, specifically Pearson's chi-squared test and variants thereof.

2. PCA Reduction – Principal component analysis (PCA) is a dimensionality reduction and machine learning method to simplify a large dataset into a smaller set while still maintaining significant patterns and trends.

3. Lasso Regression – Also known as L1 regularization – is a form of regularization for linear regression models. Regularization is a statistical method to reduce errors caused by overfitting on training data. This approach can be reflected with this formula:

$$\text{w-hat} = \text{argmin}_w \, MSE(W\,) + \|w\|_1$$

4. Recursive Feature Elimination – RFE is a feature selection method to identify a dataset’s key features. The process involves developing a model with the remaining features after repeatedly removing the least significant parts until the desired number of features is obtained. Although RFE can be used with any supervised learning method, Support Vector Machines(SVM) are the most popular pairing.

5. Logistic Regression - Logistic regression is a binary classification technique and it is an approach for prediction tasks. The core of this method is the logistic function that is a sigmoid function (an S-shaped curve). Through this function the logistic regression maps a weighted linear combination of features into real values between 0 and 1. These real values can be interpreted as probabilities: rather than predicting a class, predicting a probability of belonging to a given class of data
6. Random Forest - Random forest is a commonly-used machine learning algorithm, trademarked by Leo Breiman and Adele Cutler, that combines the output of multiple decision trees to reach a single result. Its ease of use and flexibility have fueled its adoption, as it handles both classification and regression problems.

7. XGBoost - XGBoost is a robust machine-learning algorithm that can help you understand your data and make better decisions. XGBoost is an implementation of

gradient-boosting decision trees. It has been used by data scientists and researchers worldwide to optimize their machine-learning models.

4.1.6 Model Evaluation:

Model evaluation is a critical step in the machine learning process, as it helps to determine the performance of the model on new, unseen data. The purpose of model evaluation is to assess the accuracy, robustness, and generalizability of the trained model. In this article, we will discuss the various methods of model evaluation in machine learning.

1. Confusion matrix: The confusion matrix is a table that summarizes the performance of a classification model. It shows the number of true positives, false positives, true negatives, and false negatives. The confusion matrix can be used to calculate various performance metrics such as accuracy, precision, recall, and F1 score.

2. Receiver Operating Characteristic (ROC) curve: The ROC curve is a graphical representation of the performance of a binary classification model. It plots the true positive rate (TPR) against the false positive rate (FPR) at various classification thresholds. The area under the ROC curve (AUC) is a performance metric that ranges from 0 to 1, with a higher value indicating better performance.

3. Precision-Recall curve: The precision-recall curve is a graphical representation of the performance of a binary classification model. It plots precision against recall at various classification thresholds. The area under the precision-recall curve (AUC-PR) is a performance metric that ranges from 0 to 1, with a higher value indicating better performance.

4. Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE): MAE and RMSE are performance metrics used for regression problems. MAE measures the average absolute difference between the predicted and actual values, while RMSE measures the square root of the average squared difference between the predicted and actual values. Lower values of MAE and RMSE indicate better performance.

5. Bias-Variance tradeoff: The bias-variance tradeoff is a fundamental concept in machine learning. Bias refers to the error introduced by approximating a real-world problem with a simpler model. Variance refers to the error introduced by the model's sensitivity to small fluctuations in the training data. A model with high bias will underfit the data, while a model with high variance will overfit the data. It is important to strike a balance between bias and variance to achieve good performance.

4.1.7 Model deployment

Model deployment is the final stage of the machine learning process, where the trained model is integrated into a production environment and made available for use. It involves preparing the model for deployment, creating an interface for user interaction, and setting up infrastructure to support the model's execution. There are several steps involved in model deployment, including:

1. Model optimization: Before deploying the model, it is important to optimize its performance. This can involve fine-tuning hyperparameters, optimizing the input data pipeline, or retraining the model with new data. The goal of optimization is to ensure that the model performs optimally in the production environment.

2. Model packaging: Once the model is optimized, it needs to be packaged in a format that can be easily deployed. The packaging format can vary depending on the deployment environment, but common formats include Docker containers or cloud-based virtual machines.

3. Model deployment infrastructure: Once the model is packaged, it needs to be deployed to a production environment. This typically involves setting up infrastructure to support the model's execution, such as servers or cloud-based computing resources.

4. Model integration: Once the model is deployed, it needs to be integrated with other systems in the production environment. This can involve connecting the model to a database or other backend systems, or integrating it with a user interface.

5. Model monitoring: Once the model is deployed, it is important to monitor its performance in the production environment. This can involve tracking key performance metrics, such as accuracy or response time, or monitoring for errors or anomalies.

6. Model updates: Over time, the model may need to be updated or retrained with new data. It is important to have a process in place for updating the model in the production environment, while minimizing disruption to the system.

There are several techniques that can be used to deploy machine learning models, including:

1. Cloud-based deployment: Cloud-based deployment involves deploying the model to a cloud-based computing environment, such as Amazon Web Services (AWS) or Google Cloud Platform (GCP). This approach provides scalability and flexibility, allowing the model to be easily scaled up or down based on demand.

2. Edge deployment: Edge deployment involves deploying the model to a device or system that is located close to the data source. This can be useful for applications where low latency is important, or where there is limited bandwidth for transmitting data to a cloud-based server.

3. Hybrid deployment: Hybrid deployment involves deploying the model to both cloud-based and edge-based environments. This can provide the benefits of both approaches, such as scalability and low latency.

4. API-based deployment: API-based deployment involves creating an application programming interface (API) that provides access to the model's predictions. This approach can be useful for integrating the model with other systems or applications.

CHAPTER 5
IMPLEMENTATION

```
In [1]: # import necessary libraries
        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.stats as stats
        %matplotlib inline
        from sklearn.preprocessing import LabelEncoder
        import seaborn as sns
        from scipy.stats import stats, boxcox
        from scipy.stats.mstats import winsorize
        from sklearn.feature_selection import SelectKBest, chi2, RFE
        from sklearn.preprocessing import MinMaxScaler, StandardScal
        from sklearn.decomposition import PCA
        from sklearn.model_selection import train_test_split
        from sklearn.linear_model import LogisticRegression, Lasso
        from sklearn.metrics import accuracy_score, precision_score
        from sklearn.metrics import recall_score, f1_score
        from sklearn.metrics import confusion_matrix, r2_score, mear
        from sklearn.ensemble import RandomForestClassifier
        import xgboost as xgb
```

```
In [2]: df = pd.read_csv('clinvar_conflicting.csv', low_memory=False
```

```
In [3]: df.head()
```

Out[3]:

	CHROM	POS	REF	ALT	AF_ESP	AF_EXAC	AF_TGP	
0	1	1168180	G	C	0.0771	0.10020	0.1066	
1	1	1470752	G	A	0.0000	0.00000	0.0000	Mec
2	1	1737942	A	G	0.0000	0.00001	0.0000	Hu
3	1	2160305	G	A	0.0000	0.00000	0.0000	Med
4	1	2160305	G	T	0.0000	0.00000	0.0000	Me

5 rows × 46 columns

```
In [4]: df['PolyPhen'].factorize()
```

```
Out[4]: (array([ 0,  0,  1, ..., -1, -1, -1]),
         Index(['benign', 'probably_damaging', 'possibly_damaging',
        'unknown'], dtype='object'))
```

```
In [5]: df['Consequence'].factorize()
```

```
Out[5]: (array([0, 0, 0, ..., 5, 5, 0]),
         Index(['missense_variant', 'missense_variant&splice_region
        _variant',
                'intron_variant', '5_prime_UTR_variant', '3_prime_U
        TR_variant',
                'synonymous_variant', 'downstream_gene_variant',
                'upstream_gene_variant', 'splice_region_variant&syn
        onymous_variant',
                'splice_region_variant&intron_variant', 'frameshift
        _variant',
                'inframe_deletion', 'inframe_insertion', 'stop_los
        t', 'stop_gained',
                'splice_acceptor_variant&coding_sequence_variant&in
        tron_variant',
                'stop_gained&splice_region_variant', 'splice_accept
        or_variant',
                'splice_donor_variant', 'start_lost', 'start_lost&5
        _prime_UTR_variant',
                'splice_donor_variant&coding_sequence_variant&intro
        n_variant',
                'protein_altering_variant', 'stop_gained&frameshift
        _variant',
                'splice_donor_variant&coding_sequence_variant',
                'splice_region_variant&5_prime_UTR_variant',
                'frameshift_variant&splice_region_variant',
                'start_lost&splice_region_variant',
                'splice_acceptor_variant&intron_variant',
                'inframe_deletion&splice_region_variant',
                'splice_acceptor_variant&coding_sequence_variant',
                'splice_donor_variant&intron_variant',
                'splice_region_variant&coding_sequence_variant&intr
        on_variant',
                'stop_gained&protein_altering_variant',
                'stop_retained_variant&3_prime_UTR_variant',
                'frameshift_variant&start_lost', 'stop_retained_var
        iant',
                'stop_lost&3_prime_UTR_variant',
                'inframe_insertion&splice_region_variant',
                'frameshift_variant&stop_lost',
                'splice_region_variant&3_prime_UTR_variant',
                'stop_gained&inframe_insertion', 'stop_gained&infra
        me_deletion',
                'frameshift_variant&stop_retained_variant',
                'intron_variant&non_coding_transcript_variant',
                'frameshift_variant&start_lost&start_retained_varia
        nt',
                'TF_binding_site_variant', 'intergenic_variant'],
               dtype='object'))
```

```
In [6]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65188 entries, 0 to 65187
Data columns (total 46 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   CHROM               65188 non-null  object 
 1   POS                 65188 non-null  int64  
 2   REF                 65188 non-null  object 
 3   ALT                 65188 non-null  object 
 4   AF_ESP              65188 non-null  float64
 5   AF_EXAC             65188 non-null  float64
 6   AF_TGP              65188 non-null  float64
 7   CLNDISDB            65188 non-null  object 
 8   CLNDISDBINCL        167 non-null    object 
 9   CLNDN               65188 non-null  object 
 10  CLNDNINCL           167 non-null    object 
 11  CLNHGVS             65188 non-null  object 
 12  CLNSIGINCL          167 non-null    object 
 13  CLNVC               65188 non-null  object 
 14  CLNVI               27659 non-null  object 
 15  MC                  64342 non-null  object 
 16  ORIGIN              65188 non-null  int64  
 17  SSR                 130 non-null    float64
 18  CLASS               65188 non-null  int64  
 19  Allele              65188 non-null  object 
 20  Consequence         65188 non-null  object 
 21  IMPACT              65188 non-null  object 
 22  SYMBOL              65172 non-null  object 
 23  Feature_type        65174 non-null  object 
 24  Feature             65174 non-null  object 
 25  BIOTYPE             65172 non-null  object 
 26  EXON                56295 non-null  object 
 27  INTRON              8803 non-null   object 
 28  cDNA_position       56304 non-null  object 
 29  CDS_position        55233 non-null  object 
 30  Protein_position    55233 non-null  object 
 31  Amino_acids         55184 non-null  object 
 32  Codons              55184 non-null  object 
 33  DISTANCE            108 non-null    float64
 34  STRAND              65174 non-null  float64
 35  BAM_EDIT            31969 non-null  object 
 36  SIFT                24836 non-null  object 
 37  PolyPhen            24796 non-null  object 
 38  MOTIF_NAME          2 non-null      object 
 39  MOTIF_POS           2 non-null      float64
 40  HIGH_INF_POS        2 non-null      object 
 41  MOTIF_SCORE_CHANGE  2 non-null      float64
 42  LoFtool             60975 non-null  float64
 43  CADD_PHRED          64096 non-null  float64
 44  CADD_RAW            64096 non-null  float64
 45  BLOSUM62            25593 non-null  float64
dtypes: float64(12), int64(3), object(31)
memory usage: 22.9+ MB
```

In [7]:
```
df.shape
```

Out[7]:
```
(65188, 46)
```

Data Cleaning

Null Values

In [8]:
```
# create a new dataframe with data type, nulls, & unique val
var_df = pd.DataFrame(columns=['variable_name', 'data_type',
          'missing_percentage', 'flag','unique_values_count'

missing_percentages = df.isnull().mean() * 100

# create variables and flag as numeric or categorial
for col in df.columns:
    data_type = df[col].dtype
    missing_percentage = missing_percentages[col]
    unique_values_count = df[col].nunique()
    if data_type == 'int64' or data_type == 'float64':
        flag = 'numeric'
    else:
        flag = 'categorical'
    # concat values obtained into a new dataframe called 'va
    var_df = pd.concat([var_df, pd.DataFrame({'variable_name
    'data_type': [data_type], 'missing_percentage': [missing
    'flag': [flag], 'unique_values_count': [unique_values_co

# sort variables by missing percentage value
var_df_sorted = var_df.sort_values(by='missing_percentage')
var_df_sorted.reset_index(drop=True, inplace=True)
var_df_sorted
```

Out[8]:

	variable_name	data_type	missing_percentage	
0	CHROM	object	0.000000	catego
1	Consequence	object	0.000000	catego
2	ORIGIN	int64	0.000000	num
3	IMPACT	object	0.000000	catego
4	CLNVC	object	0.000000	catego
5	CLNHGVS	object	0.000000	catego
6	CLASS	int64	0.000000	num
7	CLNDN	object	0.000000	catego
8	AF_TGP	float64	0.000000	num
9	AF_EXAC	float64	0.000000	num
10	AF_ESP	float64	0.000000	num
11	ALT	object	0.000000	catego
12	REF	object	0.000000	catego
13	POS	int64	0.000000	num
14	CLNDISDB	object	0.000000	catego
15	Allele	object	0.000000	catego
16	STRAND	float64	0.021476	num
17	Feature	object	0.021476	catego
18	Feature_type	object	0.021476	catego
19	BIOTYPE	object	0.024544	catego
20	SYMBOL	object	0.024544	catego
21	MC	object	1.297785	catego
22	CADD_RAW	float64	1.675155	num
23	CADD_PHRED	float64	1.675155	num
24	LoFtool	float64	6.462846	num
25	cDNA_position	object	13.628275	catego
26	EXON	object	13.642081	catego
27	CDS_position	object	15.271216	catego
28	Protein_position	object	15.271216	catego

	variable_name	data_type	missing_percentage	
29	Amino_acids	object	15.346383	catego
30	Codons	object	15.346383	catego
31	BAM_EDIT	object	50.958765	catego
32	CLNVI	object	57.570412	catego
33	BLOSUM62	float64	60.739707	num
34	SIFT	object	61.900963	catego
35	PolyPhen	object	61.962324	catego
36	INTRON	object	86.495981	catego
37	CLNDISDBINCL	object	99.743818	catego
38	CLNDNINCL	object	99.743818	catego
39	CLNSIGINCL	object	99.743818	catego
40	SSR	float64	99.800577	num
41	DISTANCE	float64	99.834325	num
42	MOTIF_NAME	object	99.996932	catego
43	MOTIF_POS	float64	99.996932	num
44	HIGH_INF_POS	object	99.996932	catego
45	MOTIF_SCORE_CHANGE	float64	99.996932	num

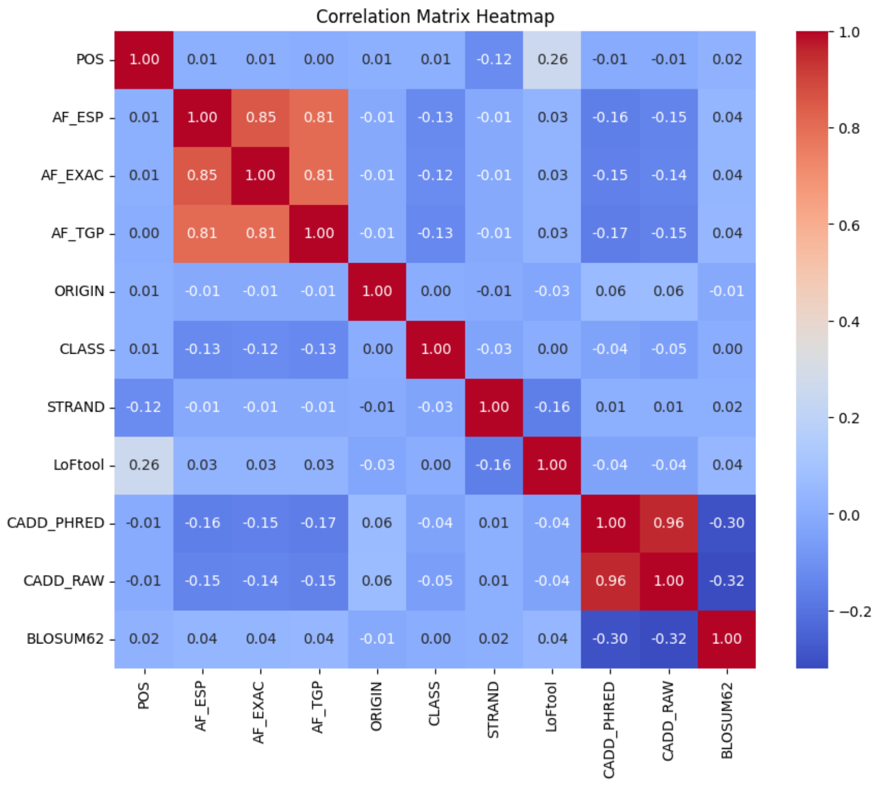
The presented chart highlights columns exhibiting an exceedingly high percentage of missing values, with nine columns surpassing a 99.5% missing threshold. These columns contain minimal non-null values, amounting to less than 0.5% of the data. To maintain the integrity and focus of the project, it is prudent to exclude columns with over 99% missing values, resulting in the removal of the nine aforementioned columns from the dataset.

```
In [9]: # Set the threshold for missing values percentage and drop i
        threshold = 99
        columns_to_drop = missing_percentages[missing_percentages >
        df = df.drop(columns=columns_to_drop)
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65188 entries, 0 to 65187
Data columns (total 37 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             65188 non-null  object 
 1   POS               65188 non-null  int64  
 2   REF               65188 non-null  object 
 3   ALT               65188 non-null  object 
 4   AF_ESP            65188 non-null  float64
 5   AF_EXAC           65188 non-null  float64
 6   AF_TGP            65188 non-null  float64
 7   CLNDISDB          65188 non-null  object 
 8   CLNDN             65188 non-null  object 
 9   CLNHGVS           65188 non-null  object 
 10  CLNVC             65188 non-null  object 
 11  CLNVI             27659 non-null  object 
 12  MC                64342 non-null  object 
 13  ORIGIN            65188 non-null  int64  
 14  CLASS             65188 non-null  int64  
 15  Allele            65188 non-null  object 
 16  Consequence       65188 non-null  object 
 17  IMPACT            65188 non-null  object 
 18  SYMBOL            65172 non-null  object 
 19  Feature_type      65174 non-null  object 
 20  Feature           65174 non-null  object 
 21  BIOTYPE           65172 non-null  object 
 22  EXON              56295 non-null  object 
 23  INTRON            8803 non-null   object 
 24  cDNA_position     56304 non-null  object 
 25  CDS_position      55233 non-null  object 
 26  Protein_position  55233 non-null  object 
 27  Amino_acids       55184 non-null  object 
 28  Codons            55184 non-null  object 
 29  STRAND            65174 non-null  float64
 30  BAM_EDIT          31969 non-null  object 
 31  SIFT              24836 non-null  object 
 32  PolyPhen          24796 non-null  object 
 33  LoFtool           60975 non-null  float64
 34  CADD_PHRED        64096 non-null  float64
 35  CADD_RAW          64096 non-null  float64
 36  BLOSUM62          25593 non-null  float64
dtypes: float64(8), int64(3), object(26)
memory usage: 18.4+ MB
```

Having removed the initial set of columns, we are now poised to delve into the exploration of variable relationships, thereby unraveling potential inconsistencies and errors embedded within the data. To gain valuable insights, a comprehensive heat matrix analysis will be employed, enabling the identification of correlations among numeric variables. This initial step promises to shed light on significant patterns and dependencies, paving the way for informed decisionmaking and further data analysis.

```
In [10]: # filter numerical columns and create correlation heat matri
         numerical_columns = df.select_dtypes(include=['int64', 'floa
         corr_matrix = numerical_columns.corr() # calculate correlati
         plt.figure(figsize=(10, 8))
         sns.heatmap(corr_matrix, cmap='coolwarm', annot=True, fmt='.
         plt.title('Correlation Matrix Heatmap')
         plt.show()
```



Drop Unnecessary Variables

Within the context of this project and experiment, there exists a selection of columns within the main 'df' dataframe that can be readily discarded due to their lack of relevance or redundancy: this analysis.

1. BAM_EDIT: Pertaining to Binary Map Alignment, it does not offer pertinent information for
2. INTRON: With a significantly low number of non-null values and containing dates, it does not align with the objectives of this study.

3. EXON: Comprising date-related information, it is not applicable since this project does not involve time series analysis.
4. CLNDISDB: Providing MedGen database identifiers, it does not contribute to the current investigation.
5. CLNHGVS: Serving as a database identifier, it does not bear relevance for the purposes of this study.
6. MC: A redundant column replicating the information found in the 'Consequence' column alongside an additional database identifier.
7. CLNVI: Representing a lab location identifier, it holds no significance in the present project.
8. SYMBOL: Constituting another identifier, it duplicates existing information and can be omitted.
9. Feature: The information it encompasses is already included in the 'Consequence' column.
10. Feature_type: The information it entails is already included in the 'Consequence' column.
11. BIOTYPE: The information it encompasses is already included in the 'Consequence' column.
12. CADD_RAW: While related to CADD_PHRED, only CADD_PHRED is essential for this genetic mutation analysis due to its usage of a more manageable scale.
13. Allele: As it contains the same values as the 'ALT' column, it can be regarded as a duplicate entry and removed accordingly. By eliminating these redundant or irrelevant columns, the data will be streamlined and more focused, enhancing the accuracy and efficiency of subsequent analyses.

In [11]:
```
df.drop(['BAM_EDIT', 'EXON', 'CLNDISDB', 'CLNHGVS',
'MC', 'CLNVI', 'SYMBOL', 'Feature', 'Feature_type',
'BIOTYPE', 'INTRON', 'CADD_RAW', 'Allele'], axis=1, inplace=
df.columns
```

Out[11]:
```
Index(['CHROM', 'POS', 'REF', 'ALT', 'AF_ESP', 'AF_EXAC',
'AF_TGP', 'CLNDN',
       'CLNVC', 'ORIGIN', 'CLASS', 'Consequence', 'IMPACT',
'cDNA_position',
       'CDS_position', 'Protein_position', 'Amino_acids',
'Codons', 'STRAND',
       'SIFT', 'PolyPhen', 'LoFtool', 'CADD_PHRED', 'BLOSUM
62'],
      dtype='object')
```

In [12]:
```
# Create an empty DataFrame to store the columns with missir
still_missing = pd.DataFrame()

# Iterate over the columns of the DataFrame and isolate miss
for column in df.columns:
    if df[column].isnull().any():
        still_missing[column] = df[column]

print("Columns with missing values:")
still_missing.info()
```

```
Columns with missing values:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65188 entries, 0 to 65187
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   cDNA_position     56304 non-null  object 
 1   CDS_position      55233 non-null  object 
 2   Protein_position  55233 non-null  object 
 3   Amino_acids       55184 non-null  object 
 4   Codons            55184 non-null  object 
 5   STRAND            65174 non-null  float64
 6   SIFT              24836 non-null  object 
 7   PolyPhen          24796 non-null  object 
 8   LoFtool           60975 non-null  float64
 9   CADD_PHRED        64096 non-null  float64
 10  BLOSUM62          25593 non-null  float64
dtypes: float64(4), object(7)
memory usage: 5.5+ MB
```

Forward Fill and Interpolation of Remaining Nulls

In order to handle missing values in the dataset, a forward fill (ffill) approach was employed for both the 'BLOSUM62' column and object columns. This method was selected to propagate the last observed non-null value forward, maintaining the temporal coherence of the data. BLOSUM62 was also transformed via forward fill because of it's low count categorical nature. For numerical columns, interpolation was utilized to estimate missing values based on existing data points, ensuring a smooth transition and preserving the underlying trends and patterns in the numerical data.

In [13]:
```
for column in still_missing.columns:
  if column == 'BLOSUM62':
    # Fill missing values in the 'BLOSUM62' column with forw
    still_missing[column] = still_missing[column].fillna(met
  elif still_missing[column].dtype == 'object':
    # Fill missing values in object columns with forward fil
    still_missing[column] = still_missing[column].fillna(met
  elif still_missing[column].dtype == 'float64':
    # Interpolate missing values in float64 columns
    still_missing[column] = still_missing[column].interpolat

# Fill missing values in 'LoFtool' column with 0
still_missing['LoFtool'] = still_missing['LoFtool'].fillna(0

# update main dataframe
df.update(still_missing)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 65188 entries, 0 to 65187
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             65188 non-null  object 
 1   POS               65188 non-null  int64  
 2   REF               65188 non-null  object 
 3   ALT               65188 non-null  object 
 4   AF_ESP            65188 non-null  float64
 5   AF_EXAC           65188 non-null  float64
 6   AF_TGP            65188 non-null  float64
 7   CLNDN             65188 non-null  object 
 8   CLNVC             65188 non-null  object 
 9   ORIGIN            65188 non-null  int64  
 10  CLASS             65188 non-null  int64  
 11  Consequence       65188 non-null  object 
 12  IMPACT            65188 non-null  object 
 13  cDNA_position     65188 non-null  object 
 14  CDS_position      65188 non-null  object 
 15  Protein_position  65188 non-null  object 
 16  Amino_acids       65188 non-null  object 
 17  Codons            65188 non-null  object 
 18  STRAND            65188 non-null  float64
 19  SIFT              65188 non-null  object 
 20  PolyPhen          65188 non-null  object 
 21  LoFtool           65188 non-null  float64
 22  CADD_PHRED        65188 non-null  float64
 23  BLOSUM62          65188 non-null  float64
dtypes: float64(7), int64(3), object(14)
memory usage: 11.9+ MB
```

Data Exploratory Analysis

Outliers and Transformations

In [14]:
```
df.nunique()
```

```
Out[14]: CHROM                    24
         POS                   63115
         REF                     866
         ALT                     458
         AF_ESP                 2842
         AF_EXAC                6667
         AF_TGP                 2087
         CLNDN                  9260
         CLNVC                     7
         ORIGIN                   31
         CLASS                     2
         Consequence              48
         IMPACT                    4
         cDNA_position         13970
         CDS_position          13663
         Protein_position       7339
         Amino_acids            1262
         Codons                 2220
         STRAND                    7
         SIFT                      4
         PolyPhen                  4
         LoFtool                5199
         CADD_PHRED            10111
         BLOSUM62                  6
         dtype: int64
```

```
In [15]: df['REF'].unique()
```

```
Out[15]:  array(['G', 'A', 'T', 'C', 'CAG', 'GCCCTCCTCTGAGTCTTCCTCCCC
          TTCCCGTA',
                 'AG', 'TTCCTCC', 'TTCC', 'CTT', 'CAGACCGCAGGCTGGAGAC
          CA',
                 'AACACCCGCAAGAAGCCGGTAGTCT', 'CCAGGCTGGGGAAG', 'TAGA
          G', 'TG', 'TC',
                 'GGAAGAA', 'ATGAGG', 'TACTA', 'AC', 'CG', 'CT', 'T
          A',
                 'AGCCGCCGCCAGCCGCAGCCATGGGCCGGGCCCGGCCGGGCCAACGCGGGC
          CGCCCAGCCCCGGCCCCGCCGCGCAGCCTCCCGCGCCACCGC',
                 'CCCCGCCGCCGCCAGCAGCCTGGGCAA', 'GA', 'CCA', 'TACG',
                 'GCCTGCACAAGGA', 'CTAAAGT', 'TGGAGGGAG', 'TT', 'CC
          T',
                 'AGACCGAAAGAAATTATCCAGGACTTGCTGGCCCATGCGGGGCTTTTTC
          C', 'TGTTG',
                 'CAAG', 'GTAGTGCC', 'AT', 'AGGTCACGGACG', 'GC', 'CA
          A',
                 'TAGCCCAGGCC', 'TCCTATTTCCCCTA', 'CCCAA', 'CAGA',
                 'GAACCCTGCAAAAAGTGACACTATC', 'ACTG', 'TTTG', 'TTTA
          A', 'AAAG',
                 'GAAGT', 'ATGAC', 'GTTGCAATGGGAGC', 'CAGTT', 'TTTT
          G', 'TAGA',
                 'GAGA', 'AGAG', 'AGAGGAG', 'CAGG', 'TGGA', 'CAGAG',
          'CAGAGAGAGAG',
                 'GTGCCCCAGGGCCAAC', 'GGT', 'AAC', 'AGCGCACCGTCTTT',
          'TAGAC', 'TGA',
                 'GAAAAA', 'CCATCAT', 'CCAT', 'GAA', 'ACTT', 'CAATAAA
          TAAATA', 'CA',
                 'TTCA', 'ATCG', 'TATC', 'CGG', 'GGGCCTCGAGGGGGAACTGG
          T',
                 'CCAGAGCCCAGGCCTCTGGCA', 'CAT', 'CACTG', 'ATC', 'TAC
          A',
                 'GGGACTGTCGACAAAGTTACGCACCCAATTGGGTCCTCCTTCGGGGTTCAG
          GGCAA', 'TAAG',
                 'CCCTT', 'CCTT', 'AAGG', 'CTG', 'TTC', 'GAAT', 'AC
          T', 'GCT',
                 'GGATGGAATT', 'AGTAGCTCT', 'ATCT', 'TCTC', 'GAG', 'C
          AGAAG',
                 'TACTC', 'TTG', 'AAAAG', 'AAT', 'ACTACGCCAAGGAGGT',
          'GCAC',
                 'CTTTT', 'AACTG', 'GTTCA', 'TGACAAAA', 'GATTCTTGGCAT
          GGCAGCTTTT',
                 'ATTGCTG', 'CATTTA', 'TTA', 'CTCT', 'TGAG', 'CGTT',
                 'GGGAGGAGGACAAAAAAGAGAAAAAGGAGAAATGTCAGGAGGAGGAGCAAG
          AGGAGCAGGAGCA',
                 'GGAGGAGCAA', 'GAGGAGC', 'TAGG', 'CACA', 'CCGAGG',
          'GTCAGGCA',
                 'CTGT', 'GGAGGAGGAGGAC', 'GGAGGAGGAAGAA', 'GGAA', 'G
          GAAGAGGAA',
                 'CGAGGAGGAAGAG', 'CGCCGCCGCCGCCGCCGCCGCTGCT', 'TGC
          T',
                 'GACCAGCTCCTT', 'TTCTC', 'TTCTCTC', 'CCCTACACCCCCTCC
          CCTGCCCCTG',
                 'CTA', 'TGCAAAAGCCGCA', 'GCC', 'ACTCTGCGCTCGCACCCAGA
```

```
GCTACCG',
       'CCCCTG', 'CATT', 'GT', 'ACTTTT', 'GAGGT', 'TATCA',
       'ATTAATTAAATATGTCATTTCATTTCTTTTTCTTTTCTT', 'GTACT',
'ACTTT',
       'CAAAT', 'CAGT', 'CAAAATAAATA', 'CAAAATA', 'AAAAT',
'AAAATAAAT',
       'AATAAATAAATATATATAT', 'TAA', 'TAAATAA', 'GCGGCTACGG
CTGCGGCTA',
       'CC', 'GGCCGCC', 'TGCTGAAAGTG', 'TCCCCGGCCGCG', 'GGC
TGC', 'CTCAT',
       'GGGGGCCGGGGCC', 'GGGGGCC', 'CGGGGCCGGGGCT', 'GGCCGG
A',
       'ACCGCGACCGGAG', 'GCA', 'CAAAG', 'CCTGGTGCTGGCG', 'C
CTGGTG',
       'TGCTGGC', 'TGCC', 'GTTC', 'CTTT', 'GAAGAA', 'TTTAAC
TTAACA',
       'TTGG', 'AGAGAGGGAGGGAAGCCATCCAGGCT', 'AAGT', 'GGCCC
GGATGAACA',
       'CCG', 'ATG', 'AGACGC', 'CCGCAT', 'TCTCA', 'TAC',
       'CCATGCCCCGTGCTTCTGGAA', 'AGG', 'CGTGTGCCCTCT', 'GGC
AGAA', 'ACTC',
       'GGA', 'ACTGT', 'TGGGCGGTGAAGCGGGCATAGA', 'TCCGC',
       'GGGAGGCGGGGACACCAGGGCCT', 'GTC', 'AGATCATG', 'AGT',
'ATTTCTTCTT',
       'ATCATCACTATAT', 'CCCT', 'TGG', 'CAATT', 'GCATT', 'T
ATCTC',
       'GTAATC', 'TCCTC', 'ACACCAC', 'GTTTAC', 'TTTC', 'TTA
A', 'TTGTG',
       'TAGT', 'CCAAGTTCG', 'CTCAG', 'GAAAAT', 'CTTTTA', 'T
AT',
       'TTTGTGCCC', 'AATTTACCAGAG', 'ATTC', 'ATACT', 'GTAAT
AAAAATTT',
       'GGTCTCTTA', 'TTCTA', 'CTCGC', 'AACAATTTTTAATGAT',
'AAAAGT',
       'ATTTTAGTT', 'TTTCTTATACAGAACAATCCCAGCC', 'TTACA',
'TCAA',
       'ATAATG', 'CTCTAGAATT', 'GCAAA', 'CTTATA', 'ATGT',
       'AAATCTGGTGACTATAC', 'TATAAG', 'GTATT', 'TGAATGGTGCA
CAG', 'GATAC',
       'ATTTCAGTGCC', 'GGTGA', 'ATTGT', 'TGAGAAACTCTC', 'TT
GAC',
       'AGAAACTGAAAG', 'TAGC', 'AACG', 'TGGGTCAGACGCGGGAAGG
C', 'CGAGA',
       'GGTGTT', 'GGTGTG', 'ACACT', 'TGTATGAAA', 'GCCT', 'A
CG', 'AGTTGCT',
       'CGT',
       'ACTGGGGGGACAGGTGTGATTCCTCAGGTTGGGGGGACAAGCATGGCTCCT
CAGGCACAGGAGACAGGTGCGGCTCCTCAGT',
       'CAGGTGTGGCTCCTCAGCCTGCGGAGAT', 'TGGCTCCTCAGGCCGGGGG
GACAGGTGC',
       'CAGGT', 'GCCTCCTCCACCT', 'ACCT', 'GAAC', 'GAAAC',
'CTTTG',
       'GCTCT', 'AGCC', 'TTGA', 'CGCAGCAGCAGCAGCA', 'TAAA',
'GATA',
        'GACTTTC', 'AAG', 'CCCGCCA', 'CTCTGAGAGCTCAGTGGAGT',
'TCA',
        'ACGCTGCCTCTTCCTGCGGGCGAGG', 'TCG',
        'CAAACCACCTTTGGATCAGATGAGCCTGAACCCAAGTCACAGCAGTCAGA
G', 'ATATC',
        'TCCAGACAC', 'CCCCTTGATGAACTT', 'GCACACGTTCTTGCAGC',
        'GATCTTTCCAATGCTGGTGGAGTGTTTGTTCACACCCCC', 'CAGTA',
'AGTT', 'GTCT',
        'AAAC', 'AAGAATT', 'TTCTTG', 'CTTTGT', 'CCAG', 'TGA
A', 'CATG',
        'AACAGTTGT', 'GCCC', 'CGA', 'ATAACAT', 'AA', 'ACAGTT
GAAAT', 'GAAA',
        'AAAT', 'TGAC', 'TAAC', 'AATG', 'CTGTTTG', 'AAAGTT',
'GACAAA',
        'TGTA', 'TGAGTCA', 'AATC', 'ATTTAAG', 'TATA', 'AAGA
T', 'CAAA',
        'AGAT', 'CTAATATG', 'TTATA', 'CTAG', 'ATGGCAGACTGACA
GT', 'TTT',
        'AAAGATTCAGGT', 'CCAAA', 'GTT',
        'GTCCATCATCAGATTTATATTCTCTGTTAACAGAAGGAAAGAGATACAGAA
TTTATCATCTTGCAACTTCAAAATCTAAAAGTAAATCTGAAAGAGCTAACATACAGTTA
GCAGCGACAAAAAAAAC',
        'CTAAAAG', 'AAACAGGTAATGCAC', 'GG', 'CGCCGCCGCT', 'A
CCG', 'GAC',
        'CAGCATAT', 'ATTAT', 'CTCTTT', 'CAAAAT', 'GGGA', 'AT
GCCCTCCTTCT',
        'ATGTGT', 'GCACCACCAC', 'CCCGCCGCCGCCCGCCCCGCAACCG',
        'ACCGCCGCCGCCGCCGCAGCAGCAGCAG', 'CGCCGCCGCCGCCGCAGCA
GCAGCA',
        'TCTTA', 'CTAA', 'TCC', 'GGACC', 'TTGTC', 'AGTTTACCA
TG', 'GAGAA',
        'TAGTC', 'ATCC', 'ACACAC', 'ACAC', 'GTA', 'AATTGCTGT
AA', 'AAGTT',
        'CAAGT', 'GTG', 'GTTGA', 'ATCTC', 'ACCCCAC', 'CACCCC
T', 'GGCC',
        'GGCCGCCGCC', 'GGGGGCCTGGCGGATCACGGGT', 'CTCTT', 'CT
TTAAG',
        'CTTCTG', 'TTAAG', 'CCAGCGGTGATAAGGCCAA', 'TCACA',
        'AGTTCTGGAGTGCTCT', 'TGGTTCCTACGAAGCTCTGAAA', 'GACT
C', 'TGAAC',
        'AATAACATAAC', 'GTAAAA', 'GTTTC', 'CGGTCAAGAGCCT',
'TGGCCTC',
        'TGAAGCGCAGGAA', 'GGCTGCTGTTGCT', 'TTGC', 'TTGCTGC',
        'TTGCTGCTGCTGCTGCTGC', 'TTGCTGCTGCTGCTGC', 'TTGCTGCT
GCTGC',
        'TTGCTGCTGC', 'GGGGCAGGGGCAAGGGCAGGGGCAGGGGCAGGGGCA
A',
        'AGGGCAGGGGCAG', 'GATGTAAGTT', 'ATCTGA', 'TGAAAG',
'CTGGGCT',
        'TCTTGGC', 'TACTG', 'ACAT', 'CTGCCAAGGA', 'TCAAA',
        'TCCCTGCAGTGCAGGAAAGGTAGGGCCGGGTGGGG', 'CCGGCTCCGCCA
CATCAAG',
        'CCGCTGGCCCGCG', 'AGC', 'GCTT', 'AAAAAGC', 'AAAAAG',
'AAAAGCA',
        'GGTT', 'GAATC', 'AAGC', 'CTTACCT', 'TCGA', 'GGTAGGT
T', 'TTTCA',
        'AGCAGGACTT', 'GATAA', 'CTGTT', 'GTTCTT', 'TCTA', 'T
AG', 'GACAA',
        'TCTG', 'CGCT', 'CGCTGCTGCT', 'TCCAGGAC', 'ATGG',
        'ATGGAACAGAAAATAACGTAAGTGTGAGGATTTTTCAACTGACTTGCAGCA
AC', 'CGGA',
        'GGGCGGC', 'ATGC', 'GCTTCCCGGGAACACC', 'CTTTTT', 'TT
TTC', 'GTGTTT',
        'TGTGGATTATCTGAAG', 'AGGAGAGGCG', 'GCCAA', 'GCTGA',
        'AAATCAATTTCCTACTGGAGATGGGTGGGAAATTGAAGTCGGTGCGAGCT
A', 'CGGGCCA',
        'GGAGCCCACCTCAGAGCCCGCCCCCAGCCCGACCACCCCA',
        'AGAGCCCACCTCAGAGCCCGCCCCCAGCCCGACCACCCCG', 'CATTTTC
AACTTACAAT',
        'GTGA', 'CTTG', 'GGCTGGTGCAGGGGCCGCCGGTGTAGGAGCT',
        'CCCCCAGCCCTCCAGGT', 'CCTGCTT', 'GCTTT', 'ATGATGCTGT
ACCAGC',
        'TCCGA', 'CCAGCAGCAGCAG', 'CCAGCAGCAG', 'CCAGCAG',
        'TCCTGGGCTGTCCGCCC', 'ATTATC', 'TTAAC', 'CAAT', 'CAC
TT', 'CTAACTT',
        'CTTGT', 'CTTGTTTGTTTGT', 'TAAAC', 'AGCCTCCC', 'GACC
TCC', 'AACC',
        'CCATA', 'TGGAATGTGGAGATCAGGAGCTCACCTTGTAAGACAAGC',
'AGA',
        'GTGGGGGATC', 'CTCGGGTCACC', 'CCACA', 'GTGGTTT', 'AA
CATCAAGTACTT',
        'TCAG', 'TGTG', 'CAACTTTCAATTGGGG', 'CACACACACACACGC
TTTTTA',
        'GATT', 'ATAAGT', 'ACAG', 'GACT', 'TTTCCTC', 'ATAC',
'CAAATTCTTT',
        'TTTT', 'AGAAC', 'GGTAAAGAACA', 'AAAAAAAAAGAAAAG',
'TGT', 'TAAAG',
        'TACTT', 'CTACTT', 'ACTGC', 'GGCATGCCATTGGGACAGCCTCA
GGTTTCT',
        'AATT', 'TGTAA', 'TTGCTTGTTCC', 'TCAGGG', 'CTTCTGCAT
GT',
        'GTATTATTACTTAAAACCAGGAAACA', 'AATAG', 'TATGG', 'TTT
GA', 'TCAC',
        'ATT', 'ATTCTT', 'GCATGCACAACAA', 'GGCATCA', 'ACGGGG
GGTGGTG',
        'ACTCC', 'TATATCACAGGCCTCGCCGA', 'AGCGGCGG', 'CGGAGG
TCCTTG',
        'TAGAAAA', 'CTGAG', 'CCACAGA', 'GTCC', 'AAAGTAT', 'A
AGAG', 'ATCTG',
        'GGACTTCTCC', 'CACAG', 'TATG', 'ATCAGGGCC', 'ATTT',
        'AGCAGCAGGCGGCTACTGCACAAGCT', 'CACAAACCT', 'ATGGGGGA
CCTGC',
        'CGAAAT', 'GTTGT', 'CGGCGCG', 'GAAGC', 'CACCATT', 'C
GGCCCTGGCCCT',
        'CCCCGGCCCGGGT', 'CTGA', 'AAGTGGCGAGTGCATCCACTCC',
'ACGAGGAAAACTG',
        'CCTGTCGCCCTGACGAATTCCAGTGCTCTGATGGAAACTGCATCCATGGCA
GCCGGCAGTGTGACCGGGAATATGACTGCAAGGACAT',
        'AACCCATC', 'AGTGCGGTGAGTCTCG', 'CTCTGC', 'GCGCTGAT
G', 'TCT',
        'TGAATGGTGTGGA', 'CTC', 'CACCCTA', 'GTGGC', 'GAT',
'CTCCTCGTCT',
        'TCTTC', 'GGATGGT', 'GGATGGTGGTGGT', 'GGATGGTGGT',
'ATGGTGGTGG',
        'GGCCCTC', 'AGTGAACAATAGCAACACACAGC', 'GAGT', 'GAC
C',
        'GTCAGTGGGGTTTGTGGCGCCCTCCC', 'CCACGGCGGC',
        'GTACCTCTGGAAGCCGCACCTCCGGCACAGCCATCTCTGGCACCTTTGGGA
GTTTCATCTCTGACACTTTGGGCAGCTC',
        'CCTCA', 'CCAAAGCCCCAGCCCTAAAAGGGGGAGCTGCGGAGCT', 'A
CGCTACCTGG',
        'CACAA', 'GCTCAGGAGGGCC', 'TGGAGGAGATGGA', 'CGGCAAGC
A',
        'AGGAGGTGAGAGGGCCG', 'CTCA', 'GGCAGCGCCA', 'TGATGA',
'CCAA',
        'GCTC', 'TAAAAAAAAAAAAA',
        'TAGGAAAACACCAGAAATTATTGTTGGCAGTTTTTGTGACTCCTCTTACTG
ATCTTCGTTCTGACTTCTCCAAGTTTCAGGAAATGATAGAAACAACTTTAGATATGGAT
C',
        'ATTAC',
        'TCAGCTTTGCTCACGTGTCAAATGGAGCACCTGTTCCATATGTACGACCAG
CCATTTTGGA',
        'ATCTCCTAGTCCC', 'GAGAATCGCA', 'GTGAAGAAGATAA', 'GGA
AAA', 'TGC',
        'GTCTCAGAACTTTGA', 'GGAACAGACTGAGA', 'GGTAA', 'TTTAA
GTCTA',
        'TGAAAA', 'CCTG', 'AGACTT', 'TTCAG', 'ATTTT', 'GTTGA
A',
        'ACTCACTACCATTCATTAGTAGAAGATTATT', 'GTAACTAACTAACTAT
AATGGAATTA',
        'TTTTTTTTTTTTT', 'ACTGTAGATG', 'TGATT',
        'ATTATAGACTGACTACATTGGAAGC', 'CATA', 'ACCAG', 'TGGAG
GA',
        'TGGAGGAGGAGGAGGAGGA', 'TGGAGGAGGAGGA', 'GGCTGGCCGTG
TTTGCC',
        'TCTGTGGGCTGGTAACAGG', 'TCTTTAAG', 'GCTCGTTTATTT',
        'ATGCTGGCTGTGCCAG', 'AATCT', 'AGTTT', 'AGCATGG', 'GT
ATA', 'CTTA',
        'CAAGCT', 'ATAAT', 'CCTTT', 'ACAATGTAGTTGCTTATTTGGCA
GC',
        'TTTGGCAGCCACCAGTA', 'ATTG', 'CCTGATAGAAGTCTTGTCT',
        'AGCCACAGTCTTTAAT', 'TTTTCCTCTTCAGGAGCAA',
        'TTTCCTCTTCTTCAGGTAGAAC', 'ATCATATTCTTCATATTCC', 'TG
GC',
        'TACAAAACAAAACAAA', 'TACAAA', 'CTTTCA', 'TTCTTTTTCAT
AC', 'TGCTTTA',
        'TCAGTATTTC', 'GTGGTGAAGAACATTCAGGCAA', 'CTATT',
        'GCGGATCCTCGGCTGC', 'ATCCTCGGCTGCCGGT', 'GATATGC',
'AATGAGT',
        'TGCA', 'TCTTTC', 'CCTCTT', 'CGAA', 'CCGCCTTCTG', 'T
AGAAAGAG',
        'TAGAA', 'CCCAGTGCCCGCAGCACGT', 'TGCCTGTCCCCTCC', 'G
```

```
GGAGAAGCC',
       'AGGTGAGCG', 'TGGGGCCACCCGGGCAGTCCCAGATCTGCGTAGGTGCG
CGC',
       'GTGGGATGTTCACAGTCAGTGACCACCCACCAGAACAGCA', 'CGAGCAG
ACTTT',
       'TCGTAAAACGTGC', 'TGTATGTTCG', 'CTCCTCAGGTTCTTGG',
'GGAATA',
       'CTGGGG', 'GACA', 'CTTTTGTTTCT', 'AACCAGTTCCAGC', 'T
GCGG',
       'GGCAGAAGA', 'GGAGTCCGGCCCGGAA', 'GCGGCTG', 'CGTTCGT
GGCAGGG',
       'GGTGAT', 'GATGCGGTCCCCACTC', 'TCAGA', 'TGGGCTTCTG',
       'GGGGTGGCAATGCA', 'GGAGGGC', 'CGAG', 'TAAGAAGAAG',
'GCAGGAGA',
       'CGTCATCCTCATA', 'ATA', 'GCAGA', 'GGGCATCATCCGGC',
'AATGAC',
       'GGCGGCGGCGGCAACTCCACC', 'TTTCACGAAGCATTTATCA', 'GCG
CTCCTGGCCT',
       'AGCTGCCGCCGCTGCC', 'TGCCGCCGCTGCCGCTGCCGCCGCCGCCGCT
GCCGCGGCC',
       'TGCCGCCGCTGCCGCTGCCGCC', 'CGCCGCCGCTGCCGCG', 'ATAAA
G', 'GAAGGTC',
       'CCTCT', 'CACAGATGCA', 'TCATACAGGTCATCGCT', 'TTCTG',
'CAAAA',
       'TCGCCGCCGC', 'TCGCCGC', 'GCCGCCGCCA', 'ATTTG', 'TTA
GA', 'TTTA',
       'AGTCTGGATGTCTTCCTCTCCTCATCCAGCTTTTACATGGCAATGACAAAG
ACTCTGTATTGTTGGGAAATTCCCGGGGCAGTAAAGAGGCTCGG',
       'CATTTAGTACTATAATATGAATTTCATGTTTGGCTTTTTTTTGCTGCCTTC
TTTTAGCCATGAGATTTCCTAATTTCTTACCTGTGTATT',
       'AACT', 'TAAAAC', 'ACAAT', 'AGAC', 'GAAATA', 'TAAAA
G', 'CCTA',
       'AACTTCT', 'TAATG', 'TAGAGTC', 'CCAGA', 'AAAAC', 'AA
ATG',
       'AACAGCCACCACTTCT', 'GGCT', 'AAGAC', 'TGAGATAA', 'GA
ACTT',
       'CAAAGA', 'TACAC', 'CCTTA', 'TAGAAATA', 'TGGACAAAAGG
CA',
       'TAGTGAGGAGGGATCTGAA', 'GCCGCTC', 'GGCA', 'GCGCGGCGG
C', 'ACAGT',
       'GGGGTCCCGCTCC', 'GTCCCGCTCCGGC', 'CTCCCGCTCGGGA',
'GGGATCTCGCTCC',
       'CAGTGGGTCCCGG', 'AAGGCCTCC', 'ACAGCAGCAGCAGCAACAGCA
G',
       'AGGCGGCGGCGGCGGCTGC', 'TAAGTTCACAATGTCACAGGACCGATTG
CCCAC',
       'CCATT', 'GATTTCTTAATGATGA', 'TGCTGCTCTGCAGAGTTGG',
'AAAAAC',
       'TTCTT', 'TAGTA', 'CCTTTTT', 'GGTGT', 'CTTATG', 'AAC
TGCATAAAATAA',
       'GATTA',
       'ACTCATGGGGAGGTAGGACACTTCAACCAACCATTTACGAAGCTTTTCTGC
CATCT', 'AGGC',
       'CGCTGCT', 'ACACGGAGT', 'AGCATCCAG', 'TTCTACTAAC',
'TGCGCAGCGC',
       'TCATA', 'CAAACAT', 'TGCCGCCGCCGCC', 'GGCCAC', 'GCA
A', 'AGATT',
       'GTTT',
       'AAAATGGAACATTTAAAGAAAGCTGACAAAATATTAATTTTGCATGAAGGT
AGCAGCTATTTTTATGGGACATTTTCAGAACTCC',
       'CAGTGAT', 'GCGGCGCCGGCGC', 'GCGGCGC', 'ATACCCTGCGGA
GGC',
       'GCGCCCGCGC', 'TAACTC', 'CTCTTA', 'TTTTTA', 'ACCACCT
G',
       'TGGAGTCTG', 'AAAACAGAC', 'ATTTGT', 'ATACCAGGTTGGT',
'CAGTG',
       'CCAGCTGTTCT', 'CGCCGCCCCCGTCGCCGCCGCCGCCA', 'CGGCCT
CCTG', 'ACCC',
       'TGGTGCA', 'AGGGGC', 'ACAGCAGCAGCAGCAG', 'ACGGCGGCG
G',
       'GCGTCGTGCACGGGT', 'CGGGTCGGGTGAGAGTGGCG', 'GGCCA',
       'AGGCTCCATGCTGCTCCCCGCCGCC', 'AAAAAAAAACAAAAAAAAAAA
C', 'CCAGT',
       'GGAGT', 'ATAT', 'ATGAGA', 'AGTA', 'TGCCGCC', 'TGCCG
CCGCC',
       'TGGCGGCGGC', 'CCCG', 'GGGACCAGCT', 'AAATT', 'GTTTCT
ATTGC',
       'GGCTGGCAGA', 'CGCACCCCGCACCCG', 'GCTCCCTCCCTCC', 'G
CTCCCTCC',
       'CGCCCT', 'GCCCT', 'CTCAA', 'TAATAAATA', 'ACCAAGG',
'TAAGGGG',
       'AGAGT', 'GGCCGCGGCGGCCGCGGCCGCGGCT', 'CTCTGCCCAAATC
A', 'ACTAT',
       'CTTCT', 'GACTTC', 'AACACAT', 'CCTTCCTCCCCTTCTT', 'A
GACAAAG',
       'AGCCATGACAAGTCGGACAGGG', 'AGGGGCCACGACAAGTCAGACC',
'CCTGA',
       'TGCAGCAGCA', 'TGCAGCAGCAGCAGCAGCA', 'TGCAGCAGCAGCAG
CA', 'TGGCGGC',
       'TGGCGGCGGCGGC', 'CCCTCCAGGACCCCCAGGA',
       'TCCTCCAGGTCCTCCTGGTCCTCAAGGACCCCCTGG',
       'CCTCTTCTCTTCTCTTCTCTTCTCTTCTCTT', 'ACAGCAACAGCAGCA
G',
       'CTCCTCCTCTTCT', 'GCTCA', 'AGGCGGCGAC', 'TTATAA', 'G
TCAA',
       'CTTTAT', 'TGTAAA', 'TTCTATACTAAAGAAATCAATCGAGTTT',
'GCTAA',
       'GGGGCTCAGGGGGGCTGGTGGGGTCCTCGGAGCTCTC',
       'GGGGGGCTGGTGGGGTCCTCGGAGCTCTCGGGCTCAGGTGGAGGT',
       'TGGTGGGGTCCTCGGAGCTCTCGGGCTCAGGTGGAGGTGGGGGCA',
       'GTGGGGTCCTCGGAGCTCTCGGGCTCAGGTGGAGGTGGGGGCAGGGGTGGG
AGCAGTGGCACGGGGGCCTT',
       'TGGGGTCCTCGGAGCTCTCGGGCTCAGGTGGAGGTGGGGGCA',
       'CGGGCTCAGGTGGAGGTGG', 'CAGGTGG', 'ATGGTGG', 'ATCACC
AT', 'TTACTC'],
      dtype=object)
```

Regarding the 'REF' and 'ALT' columns denoting the original allele and altered allele (mutation), it is customary for these values to consist of a single letter representing one of the four nucleotide bases: G, C, A, T. However, upon examination using the unique counts function, it becomes evident that these columns contain an excessive number of unique values. This can be attributed to anomalous entries where multiple letters are present instead of a single letter. These anomalies represent a very small portion of the data. Consequently, rows containing such multi-letter values will be excluded from further analysis to ensure data integrity and adherence to the expected format.

In [16]:
```
# Define the allowed values
allowed_values = ['A', 'T', 'C', 'G']

# Iterate over each column and isolate allowed values
for column in ['REF', 'ALT']:
    df = df[df[column].isin(allowed_values)]
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             61281 non-null  object 
 1   POS               61281 non-null  int64  
 2   REF               61281 non-null  object 
 3   ALT               61281 non-null  object 
 4   AF_ESP            61281 non-null  float64
 5   AF_EXAC           61281 non-null  float64
 6   AF_TGP            61281 non-null  float64
 7   CLNDN             61281 non-null  object 
 8   CLNVC             61281 non-null  object 
 9   ORIGIN            61281 non-null  int64  
 10  CLASS             61281 non-null  int64  
 11  Consequence       61281 non-null  object 
 12  IMPACT            61281 non-null  object 
 13  cDNA_position     61281 non-null  object 
 14  CDS_position      61281 non-null  object 
 15  Protein_position  61281 non-null  object 
 16  Amino_acids       61281 non-null  object 
 17  Codons            61281 non-null  object 
 18  STRAND            61281 non-null  float64
 19  SIFT              61281 non-null  object 
 20  PolyPhen          61281 non-null  object 
 21  LoFtool           61281 non-null  float64
 22  CADD_PHRED        61281 non-null  float64
 23  BLOSUM62          61281 non-null  float64
dtypes: float64(7), int64(3), object(14)
memory usage: 11.7+ MB
```

```
In [17]: df.nunique()
```

Out[17]:

```
CHROM                   24
POS                  59822
REF                      4
ALT                      4
AF_ESP                2827
AF_EXAC               6568
AF_TGP                2064
CLNDN                 8726
CLNVC                    1
ORIGIN                  30
CLASS                    2
Consequence             23
IMPACT                   4
cDNA_position        12010
CDS_position         11801
Protein_position      6381
Amino_acids            305
Codons                 766
STRAND                   7
SIFT                     4
PolyPhen                 4
LoFtool               4940
CADD_PHRED            9225
BLOSUM62                 6
dtype: int64
```

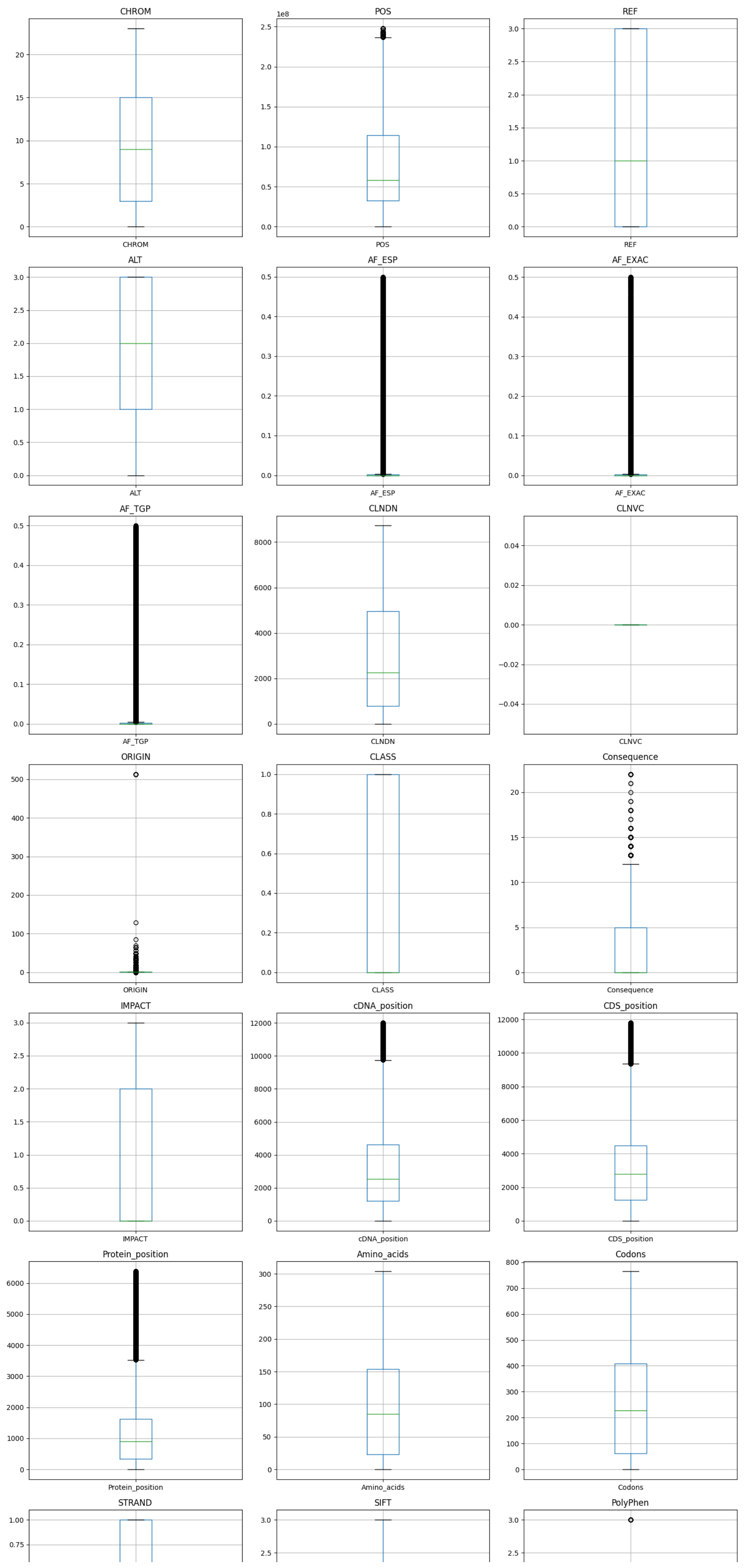
In [18]:

```
# Create a copy of df to preserve the cleaned dataframe
# before encoding
df_original = df.copy()
df_factorized = df.copy()

# Iterate over each column in the dataframe
for column in df_factorized.columns:
  # Check if the column's dtype is 'object'
  if df_factorized[column].dtype == 'object':
    df_factorized[column] = pd.factorize(df_factorized[colum

df = df_factorized
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             61281 non-null  int64  
 1   POS               61281 non-null  int64  
 2   REF               61281 non-null  int64  
 3   ALT               61281 non-null  int64  
 4   AF_ESP            61281 non-null  float64
 5   AF_EXAC           61281 non-null  float64
 6   AF_TGP            61281 non-null  float64
 7   CLNDN             61281 non-null  int64  
 8   CLNVC             61281 non-null  int64  
 9   ORIGIN            61281 non-null  int64  
 10  CLASS             61281 non-null  int64  
 11  Consequence       61281 non-null  int64  
 12  IMPACT            61281 non-null  int64  
 13  cDNA_position     61281 non-null  int64  
 14  CDS_position      61281 non-null  int64  
 15  Protein_position  61281 non-null  int64  
 16  Amino_acids       61281 non-null  int64  
 17  Codons            61281 non-null  int64  
 18  STRAND            61281 non-null  float64
 19  SIFT              61281 non-null  int64  
 20  PolyPhen          61281 non-null  int64  
 21  LoFtool           61281 non-null  float64
 22  CADD_PHRED        61281 non-null  float64
 23  BLOSUM62          61281 non-null  float64
dtypes: float64(7), int64(17)
memory usage: 11.7 MB
```

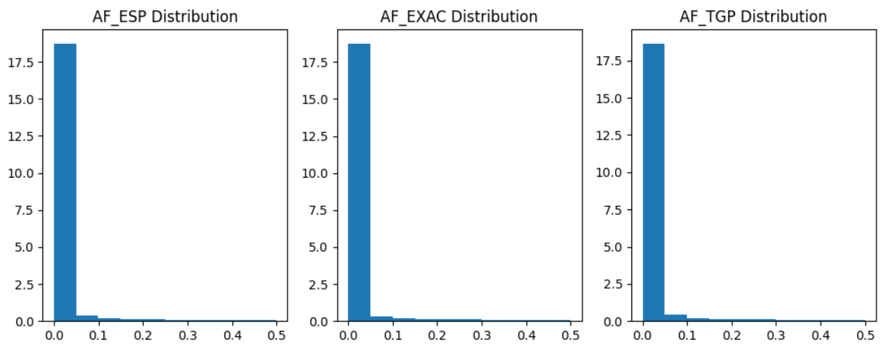
Reason for Factorized Encoding: This type of encoding was used to assign a numerical value to every unique value in the object columns without introducing any sort of rank or compromising dimensionality. One hot encoding was not chosen because it significantly increased the dimensionality of the large dataset, reducing efficiency. Additionally, label encoding was not chosen because it introduces rank into the unique values of the object columns. Currently, all object columns are considered nominal and shall be treated as such.

In [19]:
```
# Calculate the number of rows and columns for subplots
num_columns = 3
num_rows = (len(df.columns) - 1) // num_columns + 1

# Create subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(15,

# Iterate through columns and plot boxplots
for i, column in enumerate(df.columns):
  row = i // num_columns
  col = i % num_columns
  ax = axes[row][col]
  df.boxplot(column=column, ax=ax)
  ax.set_title(column)

# Adjust layout and display the plots
plt.tight_layout()
plt.show()
```


CHROM
POS
1e8
REF
ALT
AF_ESP
AF_EXAC
AF_TGP
CLNDN
CLNVC
ORIGIN
CLASS
Consequence
IMPACT
cDNA_position
CDS_position
Protein_position
Amino_acids
Codons
STRAND
SIFT
PolyPhen

Allele Frequency Visualization

```
In [20]: fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(10, 4))

         # Plot histograms for each column
         ax1.hist(df['AF_ESP'], bins=10, density=True)
         ax1.set_title('AF_ESP Distribution')

         ax2.hist(df['AF_EXAC'], bins=10, density=True)
         ax2.set_title('AF_EXAC Distribution')

         ax3.hist(df['AF_TGP'], bins=10, density=True)
         ax3.set_title('AF_TGP Distribution')

         plt.tight_layout()
         plt.show()
```



The distributions of allele frequencies displayed above exhibit similar patterns and significant right-skewness. Additionally, the absence of missing values (0% missing) and high correlation among all three columns further suggest their similarity. However, a crucial aspect requires attention. Despite the columns appearing identical and having no missing values, it is important to acknowledge that in the original database, missing values are labeled as "0" in these columns. Consequently, when iterating through the column, Python interprets these "0" values as non-null rather than null. To gain deeper insights into the actual missing values within the columns labeled as "0," let us conduct a thorough exploration.

```
In [21]:
allele_df = df[['AF_ESP', 'AF_EXAC', 'AF_TGP']]
allele_df.info()

# initiate count of three new variables
esp_zeros = 0
exac_zeros = 0
tgp_zeros = 0

# iterate through allele_df and print count of zeroes
for column in allele_df.columns:
  column_values = allele_df[column].values
  zeros_count = len(column_values[column_values == 0])
  if column == 'AF_ESP':
    esp_zeros += zeros_count
  elif column == 'AF_EXAC':
    exac_zeros += zeros_count
  elif column == 'AF_TGP':
    tgp_zeros += zeros_count
print("Count of zeroes (missing values) in AF_ESP column:",
print("Count of zeroes (missing values) in AF_EXAC column:",
print("Count of zeroes (missing values) in AF_TGP column:",
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   AF_ESP   61281 non-null  float64
 1   AF_EXAC  61281 non-null  float64
 2   AF_TGP   61281 non-null  float64
dtypes: float64(3)
memory usage: 1.9 MB
Count of zeroes (missing values) in AF_ESP column: 32042
Count of zeroes (missing values) in AF_EXAC column: 20858
Count of zeroes (missing values) in AF_TGP column: 34310
```

```
In [22]:
esp_missing = round((esp_zeros / len(allele_df)) * 100, 2)
exac_missing = round((exac_zeros / len(allele_df)) * 100, 2
tgp_missing = round((tgp_zeros / len(allele_df)) * 100, 2)
print("Percentage of actual missing values in AF_ESP column:
print("Percentage of actual missing values in AF_EXAC colum
print("Percentage of actual missing values in AF_TGP column:
```

```
Percentage of actual missing values in AF_ESP column: 52.29
Percentage of actual missing values in AF_EXAC column: 34.04
Percentage of actual missing values in AF_TGP column: 55.99
```

The subsequent step involves visualizing the distributions after removing the null values, which will provide further insights into the data.

In [23]:
```
for column in allele_df.columns:
  allele_df.loc[allele_df[column] == 0, column] = None

allele_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype  
---  ------   --------------  -----  
 0   AF_ESP   29239 non-null  float64
 1   AF_EXAC  40423 non-null  float64
 2   AF_TGP   26971 non-null  float64
dtypes: float64(3)
memory usage: 1.9 MB
```

The 'allele_df' dataframe above now has the correct count of non-null values in each column.

In [24]:
```
# Define the methods for filling null values
methods = ['mean', 'median', 'interpolation']

# Calculate skewness and kurtosis for original DataFrame
original_skewness = allele_df.skew()
original_kurtosis = allele_df.kurtosis()

# Create a dictionary to store the results
results = {}

# Iterate over the methods
for method in methods:

  # Fill null values using the respective method
  if method == 'mean':
    filled_df = allele_df.fillna(allele_df.mean())
  elif method == 'median':
    filled_df = allele_df.fillna(allele_df.median())
  elif method == 'interpolation':
    filled_df = allele_df.interpolate()

  # Calculate skewness and kurtosis
  skewness = filled_df.skew()
  kurtosis = filled_df.kurtosis()

  # Store the results in the dictionary
  results[method] = {'skewness': skewness, 'kurtosis': kurt

# Print the results
```

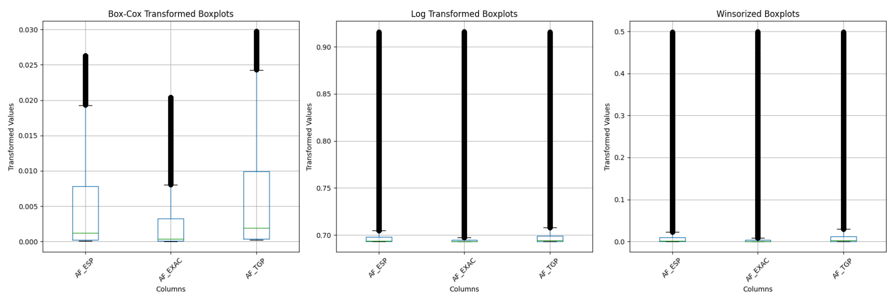
```
for method in methods:
  print(f"DataFrame with {method.capitalize()}-Filled Null V
  print("Skewness:\n", results[method]['skewness'])
  print("Kurtosis:\n", results[method]['kurtosis'])
  print()
```

```
DataFrame with Mean-Filled Null Values:
Skewness:
 AF_ESP     5.075619
AF_EXAC    5.203414
AF_TGP     4.909067
dtype: float64
Kurtosis:
 AF_ESP     29.240401
AF_EXAC    29.424542
AF_TGP     27.715173
dtype: float64

DataFrame with Median-Filled Null Values:
Skewness:
 AF_ESP     5.339052
AF_EXAC    5.328660
AF_TGP     5.227334
dtype: float64
Kurtosis:
 AF_ESP     30.488536
AF_EXAC    30.064269
AF_TGP     29.190748
dtype: float64

DataFrame with Interpolation-Filled Null Values:
Skewness:
 AF_ESP     4.117839
AF_EXAC    4.631401
AF_TGP     3.797907
dtype: float64
Kurtosis:
 AF_ESP     18.463492
AF_EXAC    22.978133
AF_TGP     15.628036
dtype: float64
```

The skewness and kurtosis values for the three types of null value filling are presented above. It has been observed that interpolation yields the least skewness and kurtosis on the data. Therefore, for the duration of the project, interpolation will be utilized for all three columns.

```
In [25]: allele_df = allele_df.interpolate()
         # Apply Box-Cox transformation to the DataFrame
         allele_df_boxcox = allele_df.copy()
         for column in allele_df_boxcox.columns:
```

```
  transformed_data, _ = boxcox(allele_df_boxcox[column].drop
  allele_df_boxcox[column].loc[~allele_df_boxcox[column].isn

log_constant = 1
allele_df_log = np.log1p(allele_df + log_constant)

# Apply winsorization to the DataFrame
allele_df_winsorized = allele_df.copy()
for column in allele_df_winsorized.columns:
  allele_df_winsorized[column] = winsorize(allele_df_winsori

# Create boxplots of transformed columns
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))

# Box-Cox transformed boxplots
allele_df_boxcox.boxplot(ax=axes[0])
axes[0].set_title("Box-Cox Transformed Boxplots")
axes[0].set_xticklabels(allele_df_boxcox.columns, rotation=4
axes[0].set_xlabel("Columns")
axes[0].set_ylabel("Transformed Values")

# Log transformed boxplots
allele_df_log.boxplot(ax=axes[1])
axes[1].set_title("Log Transformed Boxplots")
axes[1].set_xticklabels(allele_df_log.columns, rotation=45)
axes[1].set_xlabel("Columns")
axes[1].set_ylabel("Transformed Values")

# Winsorized boxplots
allele_df_winsorized.boxplot(ax=axes[2])
axes[2].set_title("Winsorized Boxplots")
axes[2].set_xticklabels(allele_df_winsorized.columns, rotati
axes[2].set_xlabel("Columns")
axes[2].set_ylabel("Transformed Values")

# Adjusting the Layout
plt.tight_layout()

# Display the plot
plt.show()
```



Considering the boxplot representations above, it becomes evident that the Box-Coxtransformation yields the most effective transformation for handling outliers in all threecolumns.

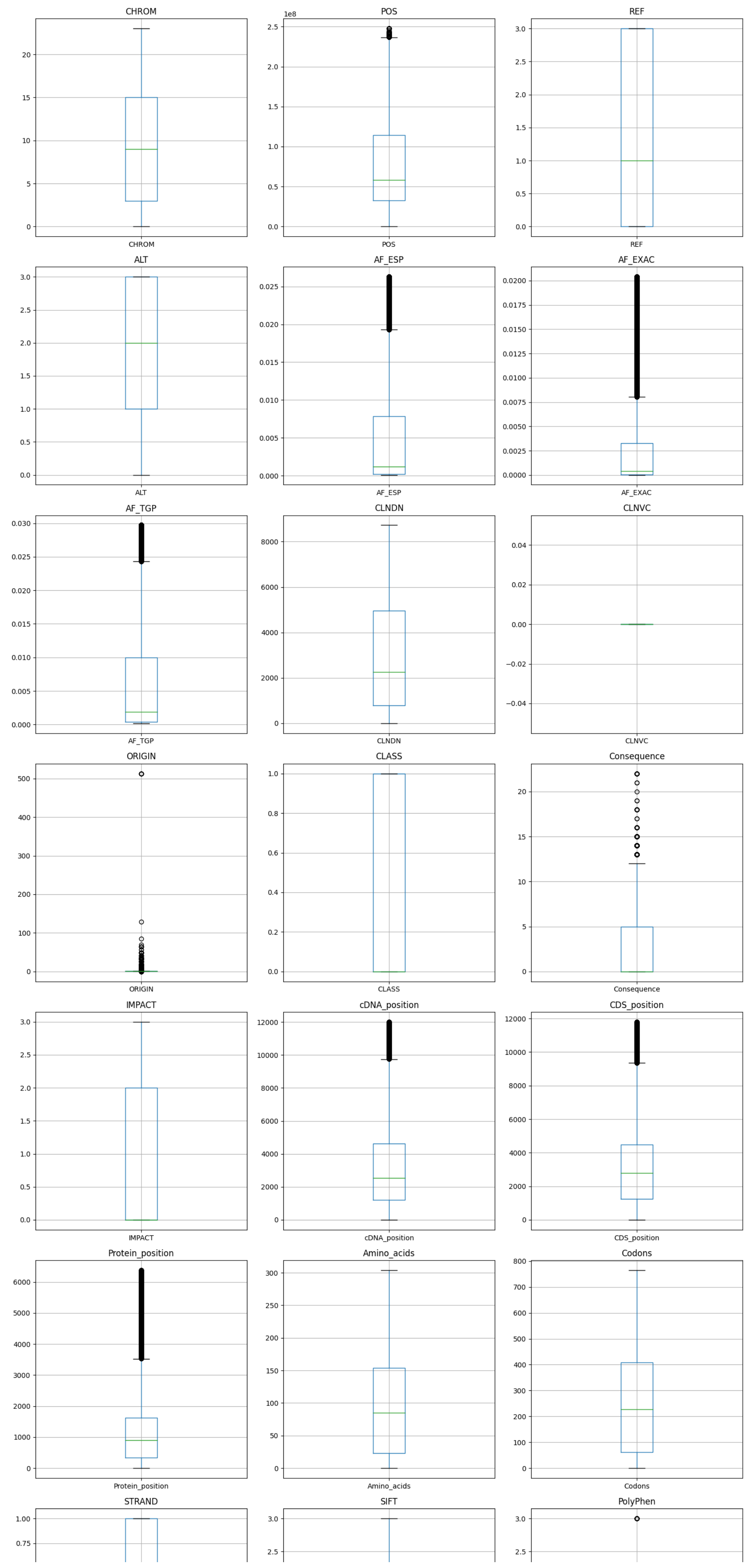
Therefore, it is appropriate to update our main dataframe accordingly to incorporatethe transformed values.

In [26]:
```
df.update(allele_df_boxcox)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             61281 non-null  int64  
 1   POS               61281 non-null  int64  
 2   REF               61281 non-null  int64  
 3   ALT               61281 non-null  int64  
 4   AF_ESP            61281 non-null  float64
 5   AF_EXAC           61281 non-null  float64
 6   AF_TGP            61281 non-null  float64
 7   CLNDN             61281 non-null  int64  
 8   CLNVC             61281 non-null  int64  
 9   ORIGIN            61281 non-null  int64  
 10  CLASS             61281 non-null  int64  
 11  Consequence       61281 non-null  int64  
 12  IMPACT            61281 non-null  int64  
 13  cDNA_position     61281 non-null  int64  
 14  CDS_position      61281 non-null  int64  
 15  Protein_position  61281 non-null  int64  
 16  Amino_acids       61281 non-null  int64  
 17  Codons            61281 non-null  int64  
 18  STRAND            61281 non-null  float64
 19  SIFT              61281 non-null  int64  
 20  PolyPhen          61281 non-null  int64  
 21  LoFtool           61281 non-null  float64
 22  CADD_PHRED        61281 non-null  float64
 23  BLOSUM62          61281 non-null  float64
dtypes: float64(7), int64(17)
memory usage: 11.7 MB
```

In [27]:
```
# Calculate the number of rows and columns for subplots
num_columns = 3
num_rows = (len(df.columns) - 1) // num_columns + 1
# Create subplots
fig, axes = plt.subplots(num_rows, num_columns, figsize=(15,
# Iterate through columns and plot boxplots
for i, column in enumerate(df.columns):
  row = i // num_columns
  col = i % num_columns
  ax = axes[row][col]
  df.boxplot(column=column, ax=ax)
  ax.set_title(column)

# Adjust layout and display the plots
```

```
plt.tight_layout()
plt.show()
```

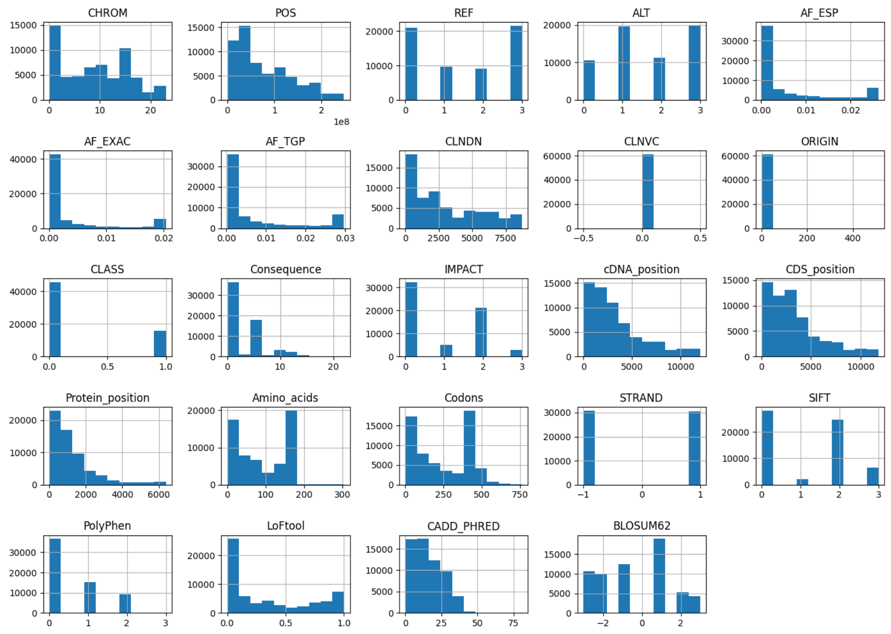

CHROM
POS
1e8
REF
ALT
AF_ESP
AF_EXAC
AF_TGP
CLNDN
CLNVC
ORIGIN
CLASS
Consequence
IMPACT
cDNA_position
CDS_position
Protein_position
Amino_acids
Codons
STRAND
SIFT
PolyPhen

```
In [28]: # Calculate z-scores for each column
         z_scores = df.apply(stats.zscore)
         # Identify outliers using z-score threshold
         z_score_threshold = 3 # Adjust the threshold as needed
         # Calculate the number of outliers for each column
         num_outliers = (np.abs(z_scores) > z_score_threshold).sum()
         # Print the number of outliers for each column
         print("Number of Outliers:")
         print(num_outliers)
```

```
Number of Outliers:
CHROM                  0
POS                    0
REF                    0
ALT                    0
AF_ESP                 0
AF_EXAC                0
AF_TGP                 0
CLNDN                  0
CLNVC                  0
ORIGIN               368
CLASS                  0
Consequence         1034
IMPACT                 0
cDNA_position        299
CDS_position         389
Protein_position    1882
Amino_acids           34
Codons                 0
STRAND                 0
SIFT                   0
PolyPhen              14
LoFtool                0
CADD_PHRED           162
BLOSUM62               0
dtype: int64
```

```
<ipython-input-28-08ae71b2dda2>:2: DeprecationWarning: Pleas
e use `zscore` from the `scipy.stats` namespace, the `scipy.
stats.stats` namespace is deprecated.
  z_scores = df.apply(stats.zscore)
```

In [29]:
```python
# Plot histograms of all columns with larger size
df.hist(figsize=(14, 10))
plt.tight_layout()
plt.show()
```



The variable 'CLNVC' contains only one unique value amongst all cells of the cleaned dataframe. Thus, it must be dropped as a duplicate variable.

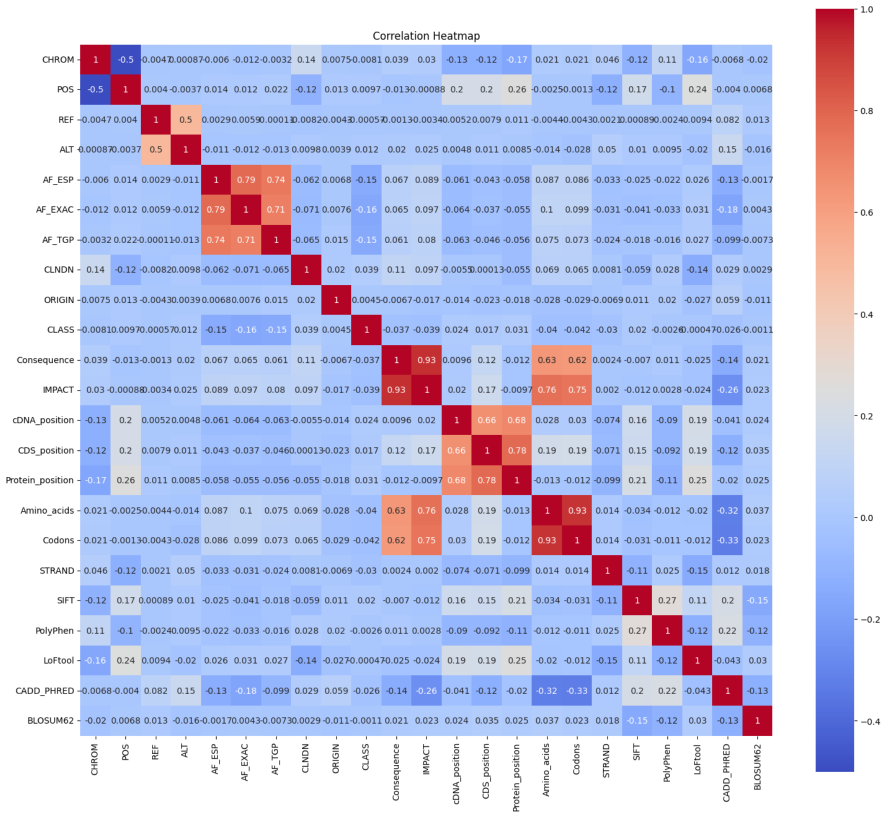
In [30]:
```python
df = df.drop('CLNVC', axis=1)
df.columns
```

Out[30]:
```
Index(['CHROM', 'POS', 'REF', 'ALT', 'AF_ESP', 'AF_EXAC',
'AF_TGP', 'CLNDN',
       'ORIGIN', 'CLASS', 'Consequence', 'IMPACT', 'cDNA_po
sition',
       'CDS_position', 'Protein_position', 'Amino_acids',
'Codons', 'STRAND',
       'SIFT', 'PolyPhen', 'LoFtool', 'CADD_PHRED', 'BLOSUM
62'],
      dtype='object')
```

Correlation Heat Map

In [31]:
```python
# Create a heatmap
plt.figure(figsize=(18, 16))
correlation_matrix = df.corr()
```

```python
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
# Set the title
plt.title('Correlation Heatmap')
# Display the heatmap
plt.show()
```



The correlation heat map above highlights the presence of high correlations among certain variables. This scenario can have both positive and negative implications for the model. When the target variable exhibits strong correlations with features, it can indicate that these variables serve as reliable predictors. However, it is important to consider the possibility of collinearity issues, where high correlations between predictor variables might lead to multicollinearity concerns.

To address these dynamics, we will delve into feature engineering and feature selection methodologies, aiming to enhance the project's analytical capabilities. By employing strategic techniques, we can uncover valuable insights, optimize the predictive power of the selected features, and mitigate the potential impact of collinearity.

In [32]:
```python
# Get the correlation matrix
corr_matrix = df.corr()
# Create a list to store correlation pairs
```

```
correlation_pairs = []
# Iterate through the correlation matrix
for i, column1 in enumerate(corr_matrix.columns):
  for j, column2 in enumerate(corr_matrix.columns):
    if i < j: # Use i < j to avoid duplicate pairs
      correlation = abs(corr_matrix[column1][column2])
      if correlation > 0.15: # Filter correlations above 0.1
        correlation_pairs.append((column1, column2, correlat
# Sort correlation pairs by the absolute correlation in desc
correlation_pairs.sort(key=lambda x: x[2], reverse=True)
print('Absolute Correlation Pairs:\n')
# Print the sorted correlation pairs
for pair in correlation_pairs:
  column1, column2, correlation = pair
  print(f"{column1} and {column2}: {correlation:.2f}")
```

```
Absolute Correlation Pairs:

Consequence and IMPACT: 0.93
Amino_acids and Codons: 0.93
AF_ESP and AF_EXAC: 0.79
CDS_position and Protein_position: 0.78
IMPACT and Amino_acids: 0.76
IMPACT and Codons: 0.75
AF_ESP and AF_TGP: 0.74
AF_EXAC and AF_TGP: 0.71
cDNA_position and Protein_position: 0.68
cDNA_position and CDS_position: 0.66
Consequence and Amino_acids: 0.63
Consequence and Codons: 0.62
CHROM and POS: 0.50
REF and ALT: 0.50
Codons and CADD_PHRED: 0.33
Amino_acids and CADD_PHRED: 0.32
SIFT and PolyPhen: 0.27
IMPACT and CADD_PHRED: 0.26
POS and Protein_position: 0.26
Protein_position and LoFtool: 0.25
POS and LoFtool: 0.24
PolyPhen and CADD_PHRED: 0.22
Protein_position and SIFT: 0.21
SIFT and CADD_PHRED: 0.20
POS and CDS_position: 0.20
POS and cDNA_position: 0.20
cDNA_position and LoFtool: 0.19
CDS_position and Codons: 0.19
CDS_position and LoFtool: 0.19
CDS_position and Amino_acids: 0.19
AF_EXAC and CADD_PHRED: 0.18
POS and SIFT: 0.17
CHROM and Protein_position: 0.17
IMPACT and CDS_position: 0.17
cDNA_position and SIFT: 0.16
AF_EXAC and CLASS: 0.16
CHROM and LoFtool: 0.16
AF_TGP and CLASS: 0.15
CDS_position and SIFT: 0.15
SIFT and BLOSUM62: 0.15
```

Feature Selection and Model Build

CLASS (pathogenicity)

In [33]:
```
X = df.drop('CLASS', axis=1)
```

In [34]:
```
y = df['CLASS']
```

In [35]:
```
# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
k = 10
selector = SelectKBest(score_func=chi2, k=k)
X_new = selector.fit_transform(X, y)
selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_[selected_indices]
selected_features = df.columns[selected_indices]
# Exclude 'CLASS' from selected features
selected_features = [feature for feature in selected_feature
                     if feature != 'CLASS']
for feature, score in zip(selected_features, selected_scores
  print(f"Feature: {feature}, Score: {score}")
```

```
Feature: AF_ESP, Score: 644.2181749822806
Feature: AF_EXAC, Score: 879.2203174205044
Feature: AF_TGP, Score: 681.2312400604937
Feature: CLNDN, Score: 23.421188968419337
Feature: Consequence, Score: 16.66052700826155
Feature: CDS_position, Score: 35.10872244011752
Feature: Protein_position, Score: 11.726915986482211
Feature: Amino_acids, Score: 14.152537035644105
Feature: Codons, Score: 18.463768253319266
```

In [36]:
```
df['AF_avg'] = (df['AF_ESP'] + df['AF_EXAC'] + df['AF_TGP'])
```

Principal Component Analysis

In [37]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
X = df[['AF_avg', 'AF_ESP', 'AF_EXAC', 'AF_TGP']]
X = StandardScaler().fit_transform(X)
sklearn_pca = PCA(n_components=1)
df["pca_1"] = sklearn_pca.fit_transform(X)
print(
'The percentage of total variance.\n',
sklearn_pca.explained_variance_ratio_
)
```

```
The percentage of total variance.
 [0.87248892]
```

In [38]:
```
df[['AF_avg', 'pca_1', 'AF_ESP', 'AF_EXAC', 'AF_TGP']].corr
```

Out[38]:

	AF_avg	pca_1	AF_ESP	AF_EXAC	AF_TGP
AF_avg	1.000000	0.999065	0.922056	0.890509	0.916935
pca_1	0.999065	1.000000	0.923808	0.908127	0.902059
AF_ESP	0.922056	0.923808	1.000000	0.786559	0.736928
AF_EXAC	0.890509	0.908127	0.786559	1.000000	0.714909
AF_TGP	0.916935	0.902059	0.736928	0.714909	1.000000

In [39]:

```
df_new = df.copy()

# Drop the specified columns
df_new.drop(['AF_ESP', 'AF_EXAC', 'AF_avg', 'AF_TGP'], axis=
X = df_new.drop('CLASS', axis=1)
y = df_new['CLASS']

# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

k = 10
selector = SelectKBest(score_func=chi2, k=k)
X_new = selector.fit_transform(X, y)

selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_[selected_indices]
selected_features = df_new.columns[selected_indices]

for feature, score in zip(selected_features, selected_scores
  print(f"Feature: {feature}, Score: {score}")
```

```
Feature: CLNDN, Score: 23.421188968419337
Feature: CLASS, Score: 16.660527008261862
Feature: Consequence, Score: 35.10872244011743
Feature: IMPACT, Score: 7.151653109976419
Feature: CDS_position, Score: 11.726915986482211
Feature: Protein_position, Score: 14.152537035644105
Feature: Amino_acids, Score: 18.463768253319266
Feature: Codons, Score: 27.61327836250723
Feature: STRAND, Score: 9.026061821254807
Feature: BLOSUM62, Score: 720.8357929510012
```

The pca variable was not selected as a feature by the Chi-Squared Test.

In [40]:

```
X = df.drop('CLASS', axis=1)

# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
```

```
k = 10
selector = SelectKBest(score_func=chi2, k=k)
X_new = selector.fit_transform(X, y)

selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_[selected_indices]
selected_features = df.columns[selected_indices]

for feature, score in zip(selected_features, selected_scores
  print(f"Feature: {feature}, Score: {score}")
```

```
Feature: AF_ESP, Score: 644.2181749822806
Feature: AF_EXAC, Score: 879.2203174205044
Feature: AF_TGP, Score: 681.2312400604937
Feature: CLNDN, Score: 23.421188968419337
Feature: CLASS, Score: 16.660527008261862
Feature: Consequence, Score: 35.10872244011743
Feature: Amino_acids, Score: 18.463768253319266
Feature: Codons, Score: 27.61327836250723
Feature: BLOSUM62, Score: 710.5914354496836
Feature: AF_avg, Score: 720.8357929510003
```

AF Statistical Significance

In [41]:
```
# Perform t-test between 'AF_avg' and 'AF_ESP'
t_statistic_es, p_value_es = stats.ttest_ind(df['AF_avg'], d
print("T-test results: AF_avg vs AF_ESP")
print(f"t-statistic: {t_statistic_es}")
print(f"p-value: {p_value_es}\n")

# Perform t-test between 'AF_avg' and 'AF_EXAC'
t_statistic_exac, p_value_exac = stats.ttest_ind(df['AF_avg'
print("T-test results: AF_avg vs AF_EXAC")
print(f"t-statistic: {t_statistic_exac}")
print(f"p-value: {p_value_exac}\n")

# Perform t-test between 'AF_avg' and 'AF_TGP'
t_statistic_tgp, p_value_tgp = stats.ttest_ind(df['AF_avg'],
print("T-test results: AF_avg vs AF_TGP")
print(f"t-statistic: {t_statistic_tgp}")
print(f"p-value: {p_value_tgp}\n")
```

```
T-test results: AF_avg vs AF_ESP
t-statistic: -7.265394688487112
p-value: 3.7418880787747494e-13

T-test results: AF_avg vs AF_EXAC
t-statistic: 50.302130265165076
p-value: 0.0

T-test results: AF_avg vs AF_TGP
t-statistic: -33.0355226075503
p-value: 3.341591294453654e-238
```

```
<ipython-input-41-58fc9d432d2b>:2: DeprecationWarning: Pleas
e use `ttest_ind` from the `scipy.stats` namespace, the `sci
py.stats.stats` namespace is deprecated.
  t_statistic_es, p_value_es = stats.ttest_ind(df['AF_avg'],
df['AF_ESP'], equal_var=False)
<ipython-input-41-58fc9d432d2b>:8: DeprecationWarning: Pleas
e use `ttest_ind` from the `scipy.stats` namespace, the `sci
py.stats.stats` namespace is deprecated.
  t_statistic_exac, p_value_exac = stats.ttest_ind(df['AF_av
g'], df['AF_EXAC'], equal_var=False)
<ipython-input-41-58fc9d432d2b>:14: DeprecationWarning: Plea
se use `ttest_ind` from the `scipy.stats` namespace, the `sc
ipy.stats.stats` namespace is deprecated.
  t_statistic_tgp, p_value_tgp = stats.ttest_ind(df['AF_av
g'], df['AF_TGP'], equal_var=False)
```

In [42]:

```
df_three = df.copy()
df_three.drop(['AF_ESP', 'AF_TGP', 'AF_EXAC'], axis=1, inpla
X = df_three.drop('CLASS', axis=1)

# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

k = 10
selector = SelectKBest(score_func=chi2, k=k)
X_new = selector.fit_transform(X, y)

selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_[selected_indices]
selected_features = df_three.columns[selected_indices]

for feature, score in zip(selected_features, selected_scores
  print(f"Feature: {feature}, Score: {score}")
```

```
Feature: CLNDN, Score: 23.421188968419337
Feature: CLASS, Score: 16.660527008261862
Feature: Consequence, Score: 35.10872244011743
Feature: CDS_position, Score: 11.726915986482146
Feature: Protein_position, Score: 14.152537035644105
Feature: Amino_acids, Score: 18.463768253319266
Feature: Codons, Score: 27.61327836250723
Feature: STRAND, Score: 9.026061821254807
Feature: BLOSUM62, Score: 710.5914354496839
Feature: AF_avg, Score: 720.8357929510012
```

Logistic Regression

In [43]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_new, y
                                                    test_size=0.2, random_st

# Create and fit the logistic regression model
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = logreg.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='micro')
recall = recall_score(y_test, y_pred, average='micro')
f1 = f1_score(y_test, y_pred, average='micro')
conf_matrix = confusion_matrix(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.7433303418454761
Precision: 0.7433303418454761
Recall: 0.7433303418454761
F1 Score: 0.7433303418454761
Confusion Matrix:
[[9111    0]
 [3146    0]]
```

Lasso Coefficients and Regression

In [44]:
```
# Define your feature matrix X and target variable y
X = df.drop('CLASS', axis=1)

# Apply feature scaling if needed
# X = StandardScaler().fit_transform(X)

# Create and fit the Lasso model
lasso = Lasso(alpha=0.1) # alpha is the regularization strer
lasso.fit(X, y)

# Get the coefficients and corresponding feature names
coefficients = lasso.coef_
feature_names = X.columns

# Print the selected features and their coefficients
selected_features = [(name, coef) for name, coef in zip(feat
                                                         coef
print("Selected Features:")
for name, coef in selected_features:
  print(name, ":", coef)
```

```
Selected Features:
CHROM : -0.0
POS : 5.947833349074799e-11
REF : 0.0
ALT : 0.0
AF_ESP : -0.0
AF_EXAC : -0.0
AF_TGP : -0.0
CLNDN : 7.213245715279824e-06
ORIGIN : 0.0
Consequence : -0.0
IMPACT : -0.0
cDNA_position : 7.317498138988874e-07
CDS_position : -1.37708864045265e-06
Protein_position : 1.0829733903382233e-05
Amino_acids : -0.0
Codons : -0.00011412032004891905
STRAND : -0.0
SIFT : 0.0
PolyPhen : 0.0
LoFtool : -0.0
CADD_PHRED : -0.0011304819800298097
BLOSUM62 : -0.0
AF_avg : -0.0
pca_1 : -0.009844368477753469
```

In [45]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                   test_size=0.2, random_
# Create and fit the Lasso model
lasso = Lasso(alpha=0.001) # alpha is the regularization str
lasso.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = lasso.predict(X_test)

# Evaluate the model performance
mse = mean_squared_error(y_test, y_pred)
```

```
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
```

```
Mean Squared Error (MSE): 0.18425791643372363
R-squared (R2): 0.03423877823918409
```

Recursive Feature Elimination (RFE)

In [46]:
```
# Separate the feature matrix X and the target variable y
X = df.drop('CLASS', axis=1)
y = df['CLASS']

# Split your data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                       test_size=0.2, random_

# Create an instance of the classifier (e.g., Logistic Regre
classifier = LogisticRegression()

# Create an instance of RFE with the classifier and 5 desire
rfe = RFE(estimator=classifier, n_features_to_select=5)

# Fit RFE on the training data
rfe.fit(X_train, y_train)

# Get the selected features from RFE
selected_features = X_train.columns[rfe.support_]

# Create a new feature matrix with the selected features
X_train_selected = X_train[selected_features]
X_test_selected = X_test[selected_features]

# Create and fit the logistic regression model using the sel
model = LogisticRegression()
model.fit(X_train_selected, y_train)

# Make predictions on the testing set
y_pred = model.predict(X_test_selected)

# Calculate the mean squared error (MSE)
mse = mean_squared_error(y_test, y_pred)

# Calculate the R-squared (R2) score
r2 = r2_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
```

```
precision = precision_score(y_test, y_pred, average='micro')
recall = recall_score(y_test, y_pred, average='micro')
f1 = f1_score(y_test, y_pred, average='micro')
conf_matrix = confusion_matrix(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("Confusion Matrix:")
print(conf_matrix)

# Print the selected features
print("Selected Features:")
for feature in selected_features:
  print(feature)
```

```
Mean Squared Error (MSE): 0.25666965815452397
R-squared (R2): -0.34529689386455953
Accuracy: 0.7433303418454761
Precision: 0.7433303418454761
Recall: 0.7433303418454761
F1 Score: 0.7433303418454761
Confusion Matrix:
[[9111    0]
 [3146    0]]
Selected Features:
POS
CLNDN
cDNA_position
CDS_position
Protein_position
```

Random Forest

In [47]:
```
# Separate the feature matrix X and the target variable y
X = df.drop('CLASS', axis=1)
y = df['CLASS']

# Split your data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                    test_size=0.2, random_st

# Create and fit the Random Forest classifier
rf_classifier = RandomForestClassifier(n_estimators=100, rar
rf_classifier.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = rf_classifier.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
```

```
# Calculate the mean squared error (MSE)
mse = mean_squared_error(y_test, y_pred)

# Calculate the R-squared (R2) score
r2 = r2_score(y_test, y_pred)

# Get the feature importances
importances = rf_classifier.feature_importances_

# Get the indices of the most important features (top k feat
k = 10
top_k_indices = importances.argsort()[-k:][::-1]

# Get the names of the selected features
selected_features = X.columns[top_k_indices]

# Print the evaluation metrics
print("Accuracy:", accuracy)
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
print("Selected Features:")

for feature in selected_features:
  print(feature)
```

```
Accuracy: 0.7549971444888635
Mean Squared Error (MSE): 0.24500285551113649
R-squared (R2): -0.28414703505253414
Selected Features:
AF_EXAC
CLNDN
POS
pca_1
AF_avg
CADD_PHRED
AF_TGP
cDNA_position
Protein_position
CDS_position
```

XGBoost

In [48]:
```
# Separate the feature matrix X and the target variable y
X = df.drop('CLASS', axis=1)
y = df['CLASS']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                  test_size=0.2, random_stat

# Create and fit the Gradient Boosting classifier with subsc
gb_classifier = xgb.XGBClassifier(n_estimators=100, random_s
gb_classifier.fit(X_train, y_train)
```

```
# Make predictions on the testing set
y_pred = gb_classifier.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)

# Calculate the mean squared error (MSE)
mse = mean_squared_error(y_test, y_pred)

# Calculate the R-squared (R2) score
r2 = r2_score(y_test, y_pred)

# Get the feature importances
importances = gb_classifier.feature_importances_

# Get the indices of the most important features (top k feat
k = 10
top_k_indices = importances.argsort()[-k:][::-1]

# Get the names of the selected features
selected_features = X.columns[top_k_indices]

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
print("Accuracy:", accuracy)
print("Selected Features:")
for feature in selected_features:
  print(feature)
```

```
Mean Squared Error (MSE): 0.2396181773680346
R-squared (R2): -0.2559240232931377
Accuracy: 0.7603818226319654
Selected Features:
IMPACT
pca_1
AF_EXAC
Consequence
CLNDN
AF_TGP
LoFtool
ORIGIN
CADD_PHRED
POS
```

```
In [49]: df_original.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 61281 entries, 0 to 65187
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   CHROM             61281 non-null  object 
 1   POS               61281 non-null  int64  
 2   REF               61281 non-null  object 
 3   ALT               61281 non-null  object 
 4   AF_ESP            61281 non-null  float64
 5   AF_EXAC           61281 non-null  float64
 6   AF_TGP            61281 non-null  float64
 7   CLNDN             61281 non-null  object 
 8   CLNVC             61281 non-null  object 
 9   ORIGIN            61281 non-null  int64  
 10  CLASS             61281 non-null  int64  
 11  Consequence       61281 non-null  object 
 12  IMPACT            61281 non-null  object 
 13  cDNA_position     61281 non-null  object 
 14  CDS_position      61281 non-null  object 
 15  Protein_position  61281 non-null  object 
 16  Amino_acids       61281 non-null  object 
 17  Codons            61281 non-null  object 
 18  STRAND            61281 non-null  float64
 19  SIFT              61281 non-null  object 
 20  PolyPhen          61281 non-null  object 
 21  LoFtool           61281 non-null  float64
 22  CADD_PHRED        61281 non-null  float64
 23  BLOSUM62          61281 non-null  float64
dtypes: float64(7), int64(3), object(14)
memory usage: 11.7+ MB
```

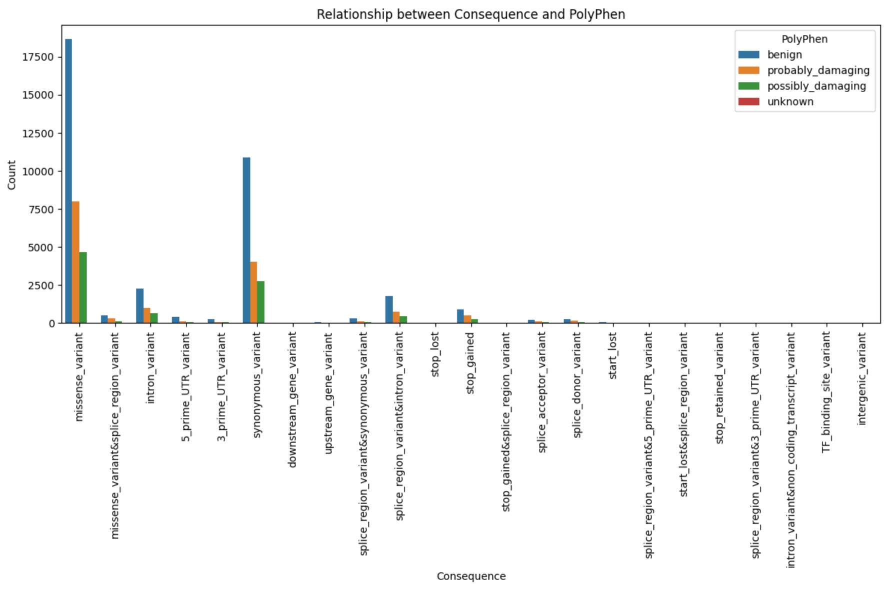
PolyPhen (Polymorphism Phenotyping) is a computational tool used in bioinformatics forpredicting the possible impact of an amino acid substitution on the structure and function of ahuman protein. This tool helps in determining whether a specific genetic mutation (variant)within a gene might be harmful (pathogenic), thus aiding in the study of disease genetics. It classifies the variants into three categories: 1. "Benign": The variant is likely harmless. 2. "Possibly Damaging": The variant could potentially be harmful, but there's someuncertainty. 3. "Probably Damaging": The variant is likely to be harmful. PolyPhen utilizes a machine learning model that incorporates a variety of protein sequence andstructural features to make these classifications, thus assisting researchers and clinicians in theinterpretation of genetic variants in humans. For this project, PolyPhen will be analyzed alongside the Consequence column. TheConsequence column provides the exact type of mutation. A significant portion of these valuesare classified as a missense variant.

```
In [50]: df_original['Consequence'].nunique()

Out[50]: 23
```

```
In [51]: # Count the unique values
         value_counts = df_original['Consequence'].value_counts()
         # Print the count of each unique value
         print(value_counts)
```

```
Consequence
missense_variant                                    31320
synonymous_variant                                  17666
intron_variant                                       3900
splice_region_variant&intron_variant                 3021
stop_gained                                          1685
missense_variant&splice_region_variant                961
5_prime_UTR_variant                                   580
splice_region_variant&synonymous_variant              552
splice_donor_variant                                  515
3_prime_UTR_variant                                   384
splice_acceptor_variant                               382
start_lost                                             92
upstream_gene_variant                                  77
stop_gained&splice_region_variant                      69
downstream_gene_variant                                22
splice_region_variant&5_prime_UTR_variant              15
intergenic_variant                                     14
stop_lost                                              10
stop_retained_variant                                   9
start_lost&splice_region_variant                        2
splice_region_variant&3_prime_UTR_variant               2
TF_binding_site_variant                                 2
intron_variant&non_coding_transcript_variant            1
Name: count, dtype: int64
```

```
In [52]: # Create a count plot
         plt.figure(figsize=(12, 8))
         sns.countplot(x='Consequence', hue='PolyPhen', data=df_origi
         plt.xlabel('Consequence')
         plt.ylabel('Count')
         plt.title('Relationship between Consequence and PolyPhen')
         plt.xticks(rotation=90) # Increase rotation value for better
         plt.legend(title='PolyPhen')
         plt.tight_layout() # Ensures labels are not cut off
         plt.show()
```

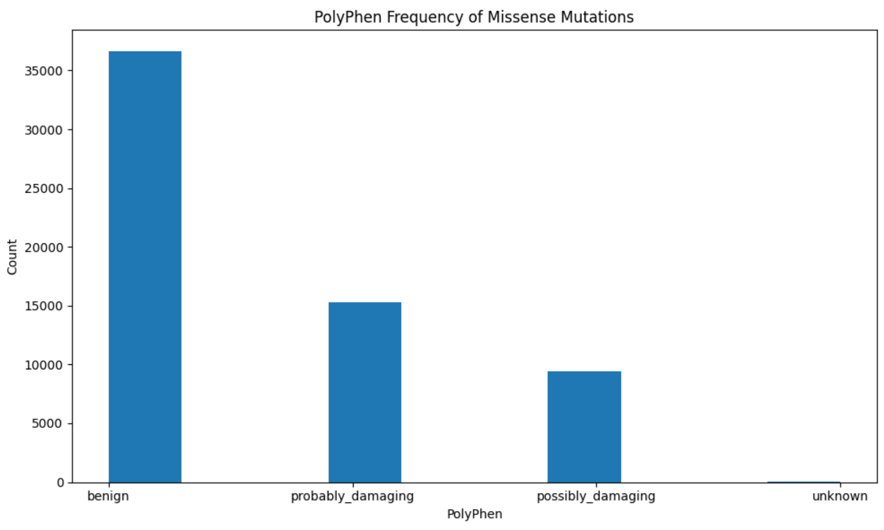


A count plot was generated above to explore the relationship between the Consequence and PolyPhen features. By visual inspection, missense variant mutations have the highest total sample count as well as the highest counts of probably and possibly damaging PoylPhen results.

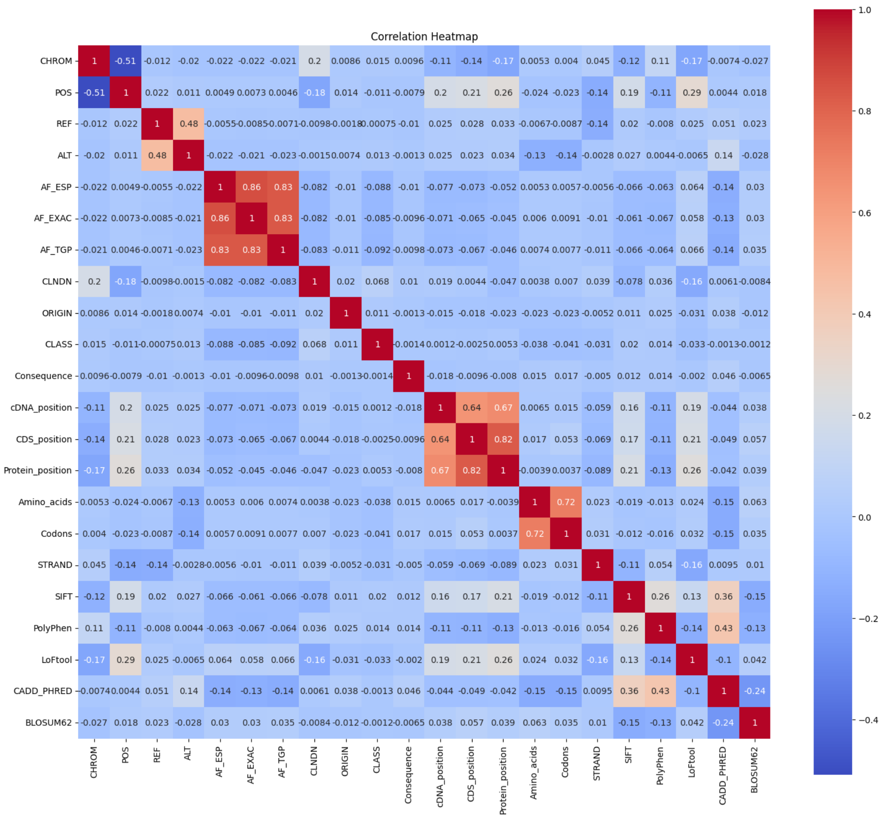
In [53]:
```
# Filter the dataframe
df_filtered = df_original[df_original['Consequence'].str.sta
```

In [54]:
```
# encode object variables for model build
df_factorized = df_filtered.copy()

# Iterate over each column in the dataframe
for column in df_factorized.columns:
  # Check if the column's dtype is 'object'
  if df_factorized[column].dtype == 'object':
    df_factorized[column] = pd.factorize(df_factorized[colum
```

In [55]:
```
# Generate bar plot for 'PolyPhen'
plt.figure(figsize=(10, 6))
plt.hist(df_original['PolyPhen'])
plt.xlabel('PolyPhen')
plt.ylabel('Count')
plt.title('PolyPhen Frequency of Missense Mutations')
plt.tight_layout()
plt.show()
```



In [56]:
```
df_factorized.drop(['IMPACT', 'CLNVC'], axis=1, inplace=True
# Create a heatmap
plt.figure(figsize=(18, 16))
correlation_matrix = df_factorized.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
plt.title('Correlation Heatmap')
plt.show()
```



In [57]:
```
X = df_factorized.drop('PolyPhen', axis=1)
y = df_factorized['PolyPhen']

# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
```

```python
X = scaler.fit_transform(X)
k = 10
selector = SelectKBest(score_func=chi2, k=k)
X_new = selector.fit_transform(X, y)

selected_indices = selector.get_support(indices=True)
selected_scores = selector.scores_[selected_indices]
selected_features = df.columns[selected_indices]

for feature, score in zip(selected_features, selected_scores
  print(f"Feature: {feature}, Score: {score}")

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_new, y
                                  test_size=0.2, random_stat

# Create and fit the logistic regression model
logreg = LogisticRegression()
logreg.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = logreg.predict(X_test)

# Calculate evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='micro')
recall = recall_score(y_test, y_pred, average='micro')

# Print the evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)

# Print the selected features
selected_features = df.columns[selected_indices]
print("Selected Features:")
for feature in selected_features:
  print(feature)
```

```
Feature: CHROM, Score: 91.83139665532157
Feature: AF_EXAC, Score: 98.37495211901782
Feature: AF_TGP, Score: 101.46518825400895
Feature: IMPACT, Score: 101.91282165167553
Feature: cDNA_position, Score: 118.85309448788595
Feature: CDS_position, Score: 146.86993896382413
Feature: STRAND, Score: 1449.1632290623747
Feature: SIFT, Score: 314.38123161415723
Feature: PolyPhen, Score: 1186.8851100003626
Feature: LoFtool, Score: 214.72414527202199
Accuracy: 0.7215425120024779
Precision: 0.7215425120024779
Recall: 0.7215425120024779
Selected Features:
CHROM
AF_EXAC
AF_TGP
IMPACT
cDNA_position
CDS_position
STRAND
SIFT
PolyPhen
LoFtool
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/linear_mode
l/_logistic.py:458: ConvergenceWarning: lbfgs failed to conv
erge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the da
ta as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.ht
ml
Please also refer to the documentation for alternative solve
r options:
    https://scikit-learn.org/stable/modules/linear_model.htm
l#logistic-regression
  n_iter_i = _check_optimize_result(
```

In [58]:

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                  test_size=0.2, random_stat

# Create and fit the Lasso model
lasso = Lasso(alpha=0.001)
lasso.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = lasso.predict(X_test)

# Evaluate the model performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

```
# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
```

```
Mean Squared Error (MSE): 0.41255609143716104
R-squared (R2): 0.24188508566910394
```

In [59]:
```
X = df_factorized.drop('PolyPhen', axis=1)
y = df_factorized['PolyPhen']
# Preprocess the feature matrix to ensure non-negative value
scaler = MinMaxScaler()
X = scaler.fit_transform(X)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                   test_size=0.2, random_stat
# Create and fit the Gradient Boosting classifier
gb_classifier = xgb.XGBClassifier(n_estimators=100, random_s
gb_classifier.fit(X_train, y_train)
# Make predictions on the testing set
y_pred_xg = gb_classifier.predict(X_test)
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred_xg)
# Calculate the mean squared error (MSE)
mse = mean_squared_error(y_test, y_pred_xg)
# Calculate the R-squared (R2) score
r2 = r2_score(y_test, y_pred_xg)
# Get the selected features
selected_features = df_factorized.drop('PolyPhen', axis=1).c
# Print the selected features
print("Selected Features:")
for feature in selected_features:
  print(feature)
# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
print("Accuracy:", accuracy)
```

```
Selected Features:
CHROM
POS
REF
ALT
AF_ESP
AF_EXAC
AF_TGP
CLNDN
ORIGIN
CLASS
Consequence
cDNA_position
CDS_position
Protein_position
Amino_acids
Codons
STRAND
SIFT
LoFtool
CADD_PHRED
BLOSUM62
Mean Squared Error (MSE): 0.43503174849001086
R-squared (R2): 0.20058371798890529
Accuracy: 0.7850394920241598
```

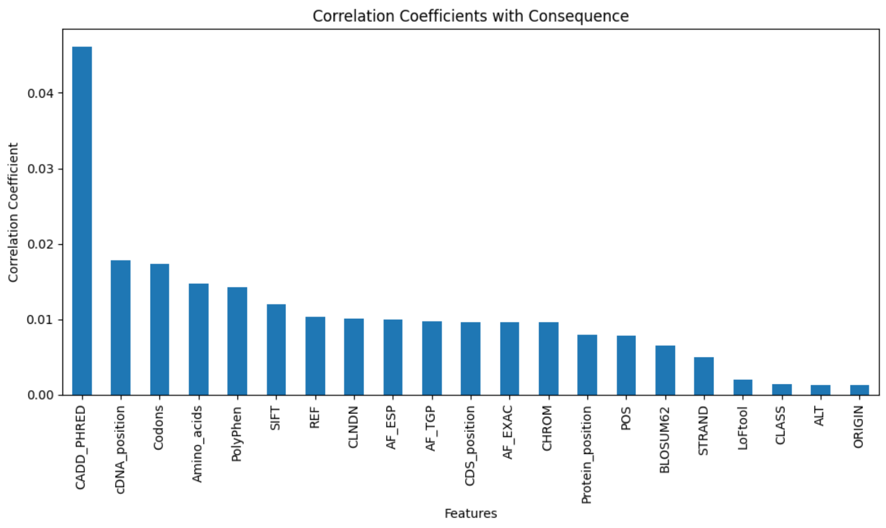
In [60]:
```
# Make predictions on the testing set
y_pred_xg = gb_classifier.predict(X_test)
# Create the confusion matrix
confusion = confusion_matrix(y_test, y_pred_xg)
print(confusion)
```

```
[[3509  165   89    0]
 [ 288 1350   94    0]
 [ 382  369  210    0]
 [   1    0    0    0]]
```

Consequence (resulting mutation)

Per the last target (PolyPhen), Consequence proves to be very important in classifying the potential damage of genetic mutations. Let's analyze the Consequence column as a whole, as opposed to strictly missense mutations.

In [61]:
```
# Calculate the correlation coefficients
correlation_matrix = df_factorized.corr()
correlation_values = correlation_matrix['Consequence'].abs(
# Sort the correlation values in descending order
sorted_correlations = correlation_values.sort_values(ascendi
# Plot the sorted correlation coefficients
plt.figure(figsize=(10, 6))
sorted_correlations = sorted_correlations.drop('Consequence'
sorted_correlations.plot(kind='bar')
plt.xlabel('Features')
plt.ylabel('Correlation Coefficient')
plt.title('Correlation Coefficients with Consequence')
plt.tight_layout()
plt.show()
```



Model Build

In [62]:
```
# Separate the feature matrix X and the target variable y
X = df.drop('Consequence', axis=1)
y = df['Consequence']
# Create and fit the Lasso regression model
lasso = Lasso(alpha=0.01) # alpha is the regularization stre
lasso.fit(X, y)
# Get the feature importance scores
lasso_coef = pd.Series(lasso.coef_, index=X.columns)
sorted_coef = lasso_coef.abs().sort_values(ascending=False)
# Print the sorted feature importance scores
print("Sorted Feature Importance Scores:")
print(sorted_coef)
```

```
Sorted Feature Importance Scores:
IMPACT              3.644115e+00
ALT                 6.994597e-02
PolyPhen            3.380137e-02
SIFT                3.137545e-02
CADD_PHRED          2.839463e-02
BLOSUM62            1.587991e-02
REF                 8.983891e-03
pca_1               7.474237e-03
Amino_acids         5.286939e-03
CHROM               2.855687e-03
Codons              1.314255e-03
ORIGIN              7.031344e-04
Protein_position    9.526619e-05
CDS_position        5.713235e-05
CLNDN               1.625339e-05
cDNA_position       6.329919e-06
POS                 4.266609e-10
CLASS               0.000000e+00
STRAND              0.000000e+00
AF_TGP              0.000000e+00
LoFtool             0.000000e+00
AF_EXAC             0.000000e+00
AF_ESP              0.000000e+00
AF_avg              0.000000e+00
dtype: float64
```

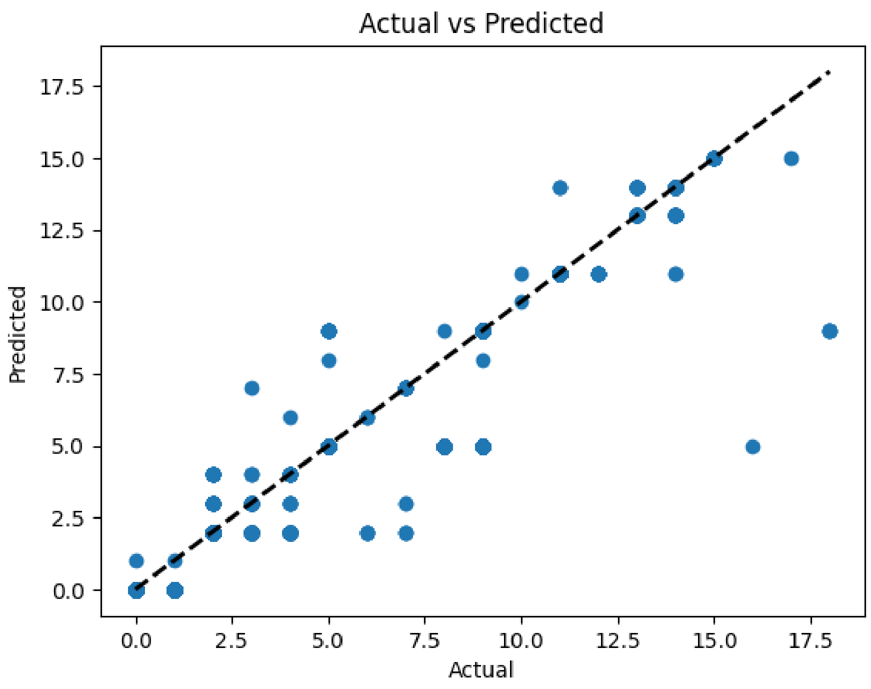
In [63]:
```
# Separate the feature matrix X and the target variable y
X = df.drop('Consequence', axis=1)
y = df['Consequence']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,\
                                test_size=0.2, random_state=

# Create and fit the XGBoost classification model
xgb_model = xgb.XGBClassifier(n_estimators=100, random_state
xgb_model.fit(X_train, y_train)

# Make predictions on the testing set
y_pred = xgb_model.predict(X_test)

# Calculate evaluation metrics
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

# Print the evaluation metrics
print("Mean Squared Error (MSE):", mse)
print("R-squared (R2):", r2)
print("Accuracy:", accuracy)

# Print the selected features
print("Selected Features:")
```

```
for feature in selected_features:
  print(feature)
```

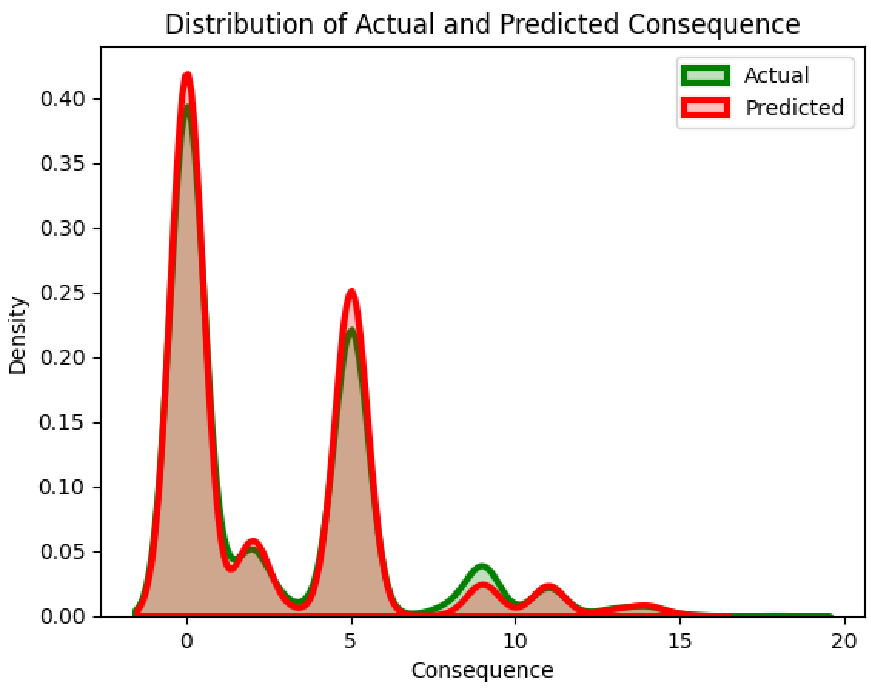
```
Mean Squared Error (MSE): 0.5107285632699682
R-squared (R2): 0.9563463820625628
Accuracy: 0.9353838622827771
Selected Features:
CHROM
POS
REF
ALT
AF_ESP
AF_EXAC
AF_TGP
CLNDN
ORIGIN
CLASS
Consequence
cDNA_position
CDS_position
Protein_position
Amino_acids
Codons
STRAND
SIFT
LoFtool
CADD_PHRED
BLOSUM62
```

Final Model Prediction Visualizations

In [64]:
```
# Create a plot of actual vs predicted values
plt.scatter(y_test, y_pred)
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted')
plt.show()
```



```
In [65]: # Create a plot of the distribution of predicted values
         sns.kdeplot(y_test, label='Actual', color='g', fill=True, li
         sns.kdeplot(y_pred, label='Predicted', color='r', fill=True,
         plt.xlabel('Consequence')
         plt.ylabel('Density')
         plt.title('Distribution of Actual and Predicted Consequence'
         plt.legend()
         plt.show()
```



CHAPTER 6
DISCUSSION AND ANALYSIS OF RESULTS

5.1 PCA - In an effort to explore the dimensionality of the data, I conducted a subset analysis focused on the three highly correlated allele frequency columns. These columns exhibit strong correlations with both the 'CLASS' variable and each other. Through Principal Component Analysis (PCA), I engineered a new variable, 'pca_1', which accounts for an impressive 87.25% of the total variance within the four allele columns. This finding suggests that a significant portion of the original variability can be effectively represented by this one-dimensional projection.

5.2 Chi-Square -The results of the chi-squared test that includes all frequency columns revealed the selection of the allele frequency columns as significant features, contrary to our previous attempt. Despite the high correlations among the allele frequencies, the 'AF_avg' column emerged as a standout feature with a high score.
Considering the persisting collinearity among the remaining allele frequency columns, we have made the definitive decision to retain only the 'AF_avg' column in order to mitigate collinearity issues. This strategic choice aims to enhance the independence of the selected features and improve the overall robustness of our model

5.3 AF Statistical Significance - The emerging findings from our data analysis underscore the divergence in allele frequency measurements across different databases, leading to potential misclassification of diseases by medical practitioners.

5.4 Logistic Regression - The logistic regression model achieved an accuracy of 0.743. However, a detailed examination of the confusion matrix, precision, recall score and F1 score revealed to all be identical to the accuracy value.

5.5 Lasso- The results indicate that the Lasso regression model might not be a good fit for the data. The relatively low R2 suggest that the model may not be capturing the underlying patterns well and is not able to explain a substantial amount of the variance in the target variable.

5.6 RFE - Recursive Feature Elimination has obtained an accuracy of 74.3%. However, all metric scores are calculated as the same again. The RFE model, similar to logistic regression, is struggling to handle the imbalanced CLASS values.

Despite applying various techniques, our models show limited success, with similar accuracy scores peaking at 0.761 However, error scores reveal that these models barely outperform mean-based predictions

XGBoost performed the best on the data and was used to build the classification model of the 'PolyPhen' target variable. Accuracy is 76%.

Accuracy: An excellent score of 93.4% signifies that the model was able to correctly predict the outcomes in the majority of the cases, thus demonstrating a strong ability to differentiate between classes in our dataset.

R-squared: The substantial value of 0.957 confirms the model's robustness, suggesting that approximately 95.7% of the variance in the dependent variable can be explained by the independent variables. This is an indicator of the model's exceptional goodness of fit.

Mean Squared Error (MSE): The low MSE of 0.505 signifies that the model's predictions are close to the actual values, demonstrating its capability to make precise predictions.

CHAPTER 7
CONCLUSION AND FUTURE SCOPE

The research explored the relationship between genetic mutation pathogenicity and disease classification using the ClinVar database. We meticulously cleaned the data, addressing missing values, outliers, and irrelevant columns. Focusing on allele frequency measures (AF_ESP, AF_EXAC, and AF_TGP), we noted null value discrepancies and potential collinearity, leading to the creation of a new feature representing their mean.

Various machine learning models were applied, including Chi-Squared, PCA Reduction, Lasso Regression, Recursive Feature Elimination, Logistic Regression, Random Forest, and XGBoost. Initial results showed an accuracy of 0.76 and an R-squared value of -0.25 due to the imbalanced 'CLASS' variable.

Analyzing the PolyPhen variable improved the XGBoost model's accuracy to 76% with an R-squared value of 0.21. Examining the 'Consequence' column further refined our understanding, achieving a 93.4% accuracy and an R-squared value of 0.957, indicating the model explained 95.7% of the variance. These findings demonstrate the complexity of genetic mutation classifications and the potential of machine learning in this field.

Further research endeavors will focus on improving genetic mutation classification models, potentially including a broader spectrum of datasets and advanced machine learning techniques. Extensive analysis of factors like "PolyPhen" and "Consequence" might enhance understanding and prediction accuracy. Real-time mutation analysis using AI and the integration of multi-omics data could yield comprehensive insights. Working with geneticists will be crucial for domain-specific feature engineering and validation. This could lead to more precise diagnoses of hereditary illnesses and customized treatment regimens.

BIBLIOGRAPHY/ REFERENCES

1. Oscar Campuzano, Georgia Sarquella-Brugada, Anna Fernandez-Falgueras, Monica Coll, Anna Iglesias, Carles Ferrer-Costa, Sergi Cesar, Elena Arbelo, Ana Garcia-Alvarez, Paloma Jorda, Rocio Toro, Coloma Tiron de Liano, Simone Grassi, Antonio Olivia, Josep Brugada, Raman Brugada(April,2020). *Reanalysis and reclassification of rare genetic variants associated with inherited arrhythmogenic syndromes*.
2. Lynn B Jorde and Stephen P Wooding(2004). *Genetic variation, classification and 'race'.*
3. Laura Valle, Eduardo Vilar, Sean V Tavtigian, Elena M Stoffel(2018). *Genetic predisposition to colorectal cancer: syndromes, genes, classification of genetic variants and implications for precision medicine.*
4. Gunnar Houge, Andreas Laner, Sebahattin Cirak, Nicole de Leeuw, Hans Scheffer and Johan T. den Dunnen(May,2021).*Stepwise ABC system for classification of any type of genetic variant.*