INTRODUCTION

According to the World Health Organization(WHO), diabetes is a chronic, metabolic disease characterized by elevated levels of blood glucose (or blood sugar), which leads over time to serious damage to the heart, blood vessels, eyes, kidneys and nerves. The most common is type 2 diabetes, usually in adults, which occurs when the body becomes resistant to insulin or doesn't make enough insulin. About 422 million people worldwide have diabetes, the majority living in low-and middle-income countries, and 1.5 million deaths are directly attributed to diabetes each year. Both the number of cases and the prevalence of diabetes have been steadily increasing over the past few decades.

Objective of study

This study aims to predict if certain patients have diabetes or not using a machine learning model built with support vector machine. The model wwas trained and tested using a PIMA Diabetes dataset. Certain risk factors and its relation to diabetes was also observed and visualizad to better show its relationship with the condition.

Data Processing

Importing the Dependencies

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.metrics import accuracy_score
```

Data Collection and Analysis

PIMA Diabetes Dataset

In [2]:
```
# loading the diabetes dataset to a pandas DataFrame
diabetes_dataset = pd.read_csv('diabetes.csv')
```

In [3]:
```
pd.read_csv?
```

```
Signature:
pd.read_csv(
    filepath_or_buffer: 'FilePath | ReadCsvBuffer[bytes] | ReadCsvBuffer[str]',
    *,
    sep: 'str | None | lib.NoDefault' = <no_default>,
    delimiter: 'str | None | lib.NoDefault' = None,
    header: "int | Sequence[int] | None | Literal['infer']" = 'infer',
    names: 'Sequence[Hashable] | None | lib.NoDefault' = <no_default>,
    index_col: 'IndexLabel | Literal[False] | None' = None,
    usecols: 'UsecolsArgType' = None,
    dtype: 'DtypeArg | None' = None,
    engine: 'CSVEngine | None' = None,
    converters: 'Mapping[Hashable, Callable] | None' = None,
    true_values: 'list | None' = None,
    false_values: 'list | None' = None,
    skipinitialspace: 'bool' = False,
    skiprows: 'list[int] | int | Callable[[Hashable], bool] | None' = None,
    skipfooter: 'int' = 0,
    nrows: 'int | None' = None,
    na_values: 'Hashable | Iterable[Hashable] | Mapping[Hashable, Iterable[Hashable]] | None' = None,
    keep_default_na: 'bool' = True,
    na_filter: 'bool' = True,
    verbose: 'bool | lib.NoDefault' = <no_default>,
    skip_blank_lines: 'bool' = True,
    parse_dates: 'bool | Sequence[Hashable] | None' = None,
    infer_datetime_format: 'bool | lib.NoDefault' = <no_default>,
    keep_date_col: 'bool | lib.NoDefault' = <no_default>,
    date_parser: 'Callable | lib.NoDefault' = <no_default>,
    date_format: 'str | dict[Hashable, str] | None' = None,
    dayfirst: 'bool' = False,
    cache_dates: 'bool' = True,
    iterator: 'bool' = False,
    chunksize: 'int | None' = None,
    compression: 'CompressionOptions' = 'infer',
    thousands: 'str | None' = None,
    decimal: 'str' = '.',
    lineterminator: 'str | None' = None,
```

```
    quotechar: 'str' = '"',
    quoting: 'int' = 0,
    doublequote: 'bool' = True,
    escapechar: 'str | None' = None,
    comment: 'str | None' = None,
    encoding: 'str | None' = None,
    encoding_errors: 'str | None' = 'strict',
    dialect: 'str | csv.Dialect | None' = None,
    on_bad_lines: 'str' = 'error',
    delim_whitespace: 'bool | lib.NoDefault' = <no_default>,
    low_memory: 'bool' = True,
    memory_map: 'bool' = False,
    float_precision: "Literal['high', 'legacy'] | None" = None,
    storage_options: 'StorageOptions | None' = None,
    dtype_backend: 'DtypeBackend | lib.NoDefault' = <no_default>,
) -> 'DataFrame | TextFileReader'
Docstring:
Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file
into chunks.

Additional help can be found in the online docs for
`IO Tools <https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html>`_.

Parameters
----------
filepath_or_buffer : str, path object or file-like object
    Any valid string path is acceptable. The string could be a URL. Valid
    URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is
    expected. A local file could be: file://localhost/path/to/table.csv.

    If you want to pass in a path object, pandas accepts any ``os.PathLike``.

    By file-like object, we refer to objects with a ``read()`` method, such as
    a file handle (e.g. via builtin ``open`` function) or ``StringIO``.
sep : str, default ','
    Character or regex pattern to treat as the delimiter. If ``sep=None``, the
    C engine cannot automatically detect
    the separator, but the Python parsing engine can, meaning the latter will
    be used and automatically detect the separator from only the first valid
    row of the file by Python's builtin sniffer tool, ``csv.Sniffer``.
    In addition, separators longer than 1 character and different from
    ``'\s+'`` will be interpreted as regular expressions and will also force
    the use of the Python parsing engine. Note that regex delimiters are prone
    to ignoring quoted data. Regex example: ``'\r\t'``.
delimiter : str, optional
    Alias for ``sep``.
header : int, Sequence of int, 'infer' or None, default 'infer'
    Row number(s) containing column labels and marking the start of the
    data (zero-indexed). Default behavior is to infer the column names: if no ``names``
    are passed the behavior is identical to ``header=0`` and column
    names are inferred from the first line of the file, if column
    names are passed explicitly to ``names`` then the behavior is identical to
    ``header=None``. Explicitly pass ``header=0`` to be able to
    replace existing names. The header can be a list of integers that
    specify row locations for a :class:`~pandas.MultiIndex` on the columns
    e.g. ``[0, 1, 3]``. Intervening rows that are not specified will be
    skipped (e.g. 2 in this example is skipped). Note that this
    parameter ignores commented lines and empty lines if
    ``skip_blank_lines=True``, so ``header=0`` denotes the first line of
    data rather than the first line of the file.
names : Sequence of Hashable, optional
    Sequence of column labels to apply. If the file contains a header row,
    then you should explicitly pass ``header=0`` to override the column names.
    Duplicates in this list are not allowed.
index_col : Hashable, Sequence of Hashable or False, optional
  Column(s) to use as row label(s), denoted either by column labels or column
  indices.  If a sequence of labels or indices is given, :class:`~pandas.MultiIndex`
  will be formed for the row labels.

  Note: ``index_col=False`` can be used to force pandas to *not* use the first
  column as the index, e.g., when you have a malformed file with delimiters at
  the end of each line.
usecols : Sequence of Hashable or Callable, optional
    Subset of columns to select, denoted either by column labels or column indices.
    If list-like, all elements must either
    be positional (i.e. integer indices into the document columns) or strings
    that correspond to column names provided either by the user in ``names`` or
    inferred from the document header row(s). If ``names`` are given, the document
    header row(s) are not taken into account. For example, a valid list-like
    ``usecols`` parameter would be ``[0, 1, 2]`` or ``['foo', 'bar', 'baz']``.
    Element order is ignored, so ``usecols=[0, 1]`` is the same as ``[1, 0]``.
```

```
    To instantiate a :class:`~pandas.DataFrame` from ``data`` with element order
    preserved use ``pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]``
    for columns in ``['foo', 'bar']`` order or
    ``pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]``
    for ``['bar', 'foo']`` order.

    If callable, the callable function will be evaluated against the column
    names, returning names where the callable function evaluates to ``True``. An
    example of a valid callable argument would be ``lambda x: x.upper() in
    ['AAA', 'BBB', 'DDD']``. Using this parameter results in much faster
    parsing time and lower memory usage.
dtype : dtype or dict of {Hashable : dtype}, optional
    Data type(s) to apply to either the whole dataset or individual columns.
    E.g., ``{'a': np.float64, 'b': np.int32, 'c': 'Int64'}``
    Use ``str`` or ``object`` together with suitable ``na_values`` settings
    to preserve and not interpret ``dtype``.
    If ``converters`` are specified, they will be applied INSTEAD
    of ``dtype`` conversion.

    .. versionadded:: 1.5.0

        Support for ``defaultdict`` was added. Specify a ``defaultdict`` as input where
        the default determines the ``dtype`` of the columns which are not explicitly
        listed.
engine : {'c', 'python', 'pyarrow'}, optional
    Parser engine to use. The C and pyarrow engines are faster, while the python engine
    is currently more feature-complete. Multithreading is currently only supported by
    the pyarrow engine.

    .. versionadded:: 1.4.0

        The 'pyarrow' engine was added as an *experimental* engine, and some features
        are unsupported, or may not work correctly, with this engine.
converters : dict of {Hashable : Callable}, optional
    Functions for converting values in specified columns. Keys can either
    be column labels or column indices.
true_values : list, optional
    Values to consider as ``True`` in addition to case-insensitive variants of 'True'.
false_values : list, optional
    Values to consider as ``False`` in addition to case-insensitive variants of 'False'.
skipinitialspace : bool, default False
    Skip spaces after delimiter.
skiprows : int, list of int or Callable, optional
    Line numbers to skip (0-indexed) or number of lines to skip (``int``)
    at the start of the file.

    If callable, the callable function will be evaluated against the row
    indices, returning ``True`` if the row should be skipped and ``False`` otherwise.
    An example of a valid callable argument would be ``lambda x: x in [0, 2]``.
skipfooter : int, default 0
    Number of lines at bottom of file to skip (Unsupported with ``engine='c'``).
nrows : int, optional
    Number of rows of file to read. Useful for reading pieces of large files.
na_values : Hashable, Iterable of Hashable or dict of {Hashable : Iterable}, optional
    Additional strings to recognize as ``NA``/``NaN``. If ``dict`` passed, specific
    per-column ``NA`` values.  By default the following values are interpreted as
    ``NaN``: " ", "#N/A", "#N/A N/A", "#NA", "-1.#IND", "-1.#QNAN", "-NaN", "-nan",
    "1.#IND", "1.#QNAN", "<NA>", "N/A", "NA", "NULL", "NaN", "None",
    "n/a", "nan", "null ".

keep_default_na : bool, default True
    Whether or not to include the default ``NaN`` values when parsing the data.
    Depending on whether ``na_values`` is passed in, the behavior is as follows:

    * If ``keep_default_na`` is ``True``, and ``na_values`` are specified, ``na_values``
      is appended to the default ``NaN`` values used for parsing.
    * If ``keep_default_na`` is ``True``, and ``na_values`` are not specified, only
      the default ``NaN`` values are used for parsing.
    * If ``keep_default_na`` is ``False``, and ``na_values`` are specified, only
      the ``NaN`` values specified ``na_values`` are used for parsing.
    * If ``keep_default_na`` is ``False``, and ``na_values`` are not specified, no
      strings will be parsed as ``NaN``.

    Note that if ``na_filter`` is passed in as ``False``, the ``keep_default_na`` and
    ``na_values`` parameters will be ignored.
na_filter : bool, default True
    Detect missing value markers (empty strings and the value of ``na_values``). In
    data without any ``NA`` values, passing ``na_filter=False`` can improve the
    performance of reading a large file.
verbose : bool, default False
    Indicate number of ``NA`` values placed in non-numeric columns.

    .. deprecated:: 2.2.0
```

```
skip_blank_lines : bool, default True
    If ``True``, skip over blank lines rather than interpreting as ``NaN`` values.
parse_dates : bool, list of Hashable, list of lists or dict of {Hashable : list}, default False
    The behavior is as follows:

    * ``bool``. If ``True`` -> try parsing the index. Note: Automatically set to
      ``True`` if ``date_format`` or ``date_parser`` arguments have been passed.
    * ``list`` of ``int`` or names. e.g. If ``[1, 2, 3]`` -> try parsing columns 1, 2, 3
      each as a separate date column.
    * ``list`` of ``list``. e.g.  If ``[[1, 3]]`` -> combine columns 1 and 3 and parse
      as a single date column. Values are joined with a space before parsing.
    * ``dict``, e.g. ``{'foo' : [1, 3]}`` -> parse columns 1, 3 as date and call
      result 'foo'. Values are joined with a space before parsing.

    If a column or index cannot be represented as an array of ``datetime``,
    say because of an unparsable value or a mixture of timezones, the column
    or index will be returned unaltered as an ``object`` data type. For
    non-standard ``datetime`` parsing, use :func:`~pandas.to_datetime` after
    :func:`~pandas.read_csv`.

    Note: A fast-path exists for iso8601-formatted dates.
infer_datetime_format : bool, default False
    If ``True`` and ``parse_dates`` is enabled, pandas will attempt to infer the
    format of the ``datetime`` strings in the columns, and if it can be inferred,
    switch to a faster method of parsing them. In some cases this can increase
    the parsing speed by 5-10x.

    .. deprecated:: 2.0.0
        A strict version of this argument is now the default, passing it has no effect.

keep_date_col : bool, default False
    If ``True`` and ``parse_dates`` specifies combining multiple columns then
    keep the original columns.
date_parser : Callable, optional
    Function to use for converting a sequence of string columns to an array of
    ``datetime`` instances. The default uses ``dateutil.parser.parser`` to do the
    conversion. pandas will try to call ``date_parser`` in three different ways,
    advancing to the next if an exception occurs: 1) Pass one or more arrays
    (as defined by ``parse_dates``) as arguments; 2) concatenate (row-wise) the
    string values from the columns defined by ``parse_dates`` into a single array
    and pass that; and 3) call ``date_parser`` once for each row using one or
    more strings (corresponding to the columns defined by ``parse_dates``) as
    arguments.

    .. deprecated:: 2.0.0
       Use ``date_format`` instead, or read in as ``object`` and then apply
       :func:`~pandas.to_datetime` as-needed.
date_format : str or dict of column -> format, optional
    Format to use for parsing dates when used in conjunction with ``parse_dates``.
    The strftime to parse time, e.g. :const:`"%d/%m/%Y"`. See
    `strftime documentation
    <https://docs.python.org/3/library/datetime.html
    #strftime-and-strptime-behavior>`_ for more information on choices, though
    note that :const:`"%f"` will parse all the way up to nanoseconds.
    You can also pass:

    - "ISO8601", to parse any `ISO8601 <https://en.wikipedia.org/wiki/ISO_8601>`_
        time string (not necessarily in exactly the same format);
    - "mixed", to infer the format for each element individually. This is risky,
        and you should probably use it along with `dayfirst`.

    .. versionadded:: 2.0.0
dayfirst : bool, default False
    DD/MM format dates, international and European format.
cache_dates : bool, default True
    If ``True``, use a cache of unique, converted dates to apply the ``datetime``
    conversion. May produce significant speed-up when parsing duplicate
    date strings, especially ones with timezone offsets.

iterator : bool, default False
    Return ``TextFileReader`` object for iteration or getting chunks with
    ``get_chunk()``.
chunksize : int, optional
    Number of lines to read from the file per chunk. Passing a value will cause the
    function to return a ``TextFileReader`` object for iteration.
    See the `IO Tools docs
    <https://pandas.pydata.org/pandas-docs/stable/io.html#io-chunking>`_
    for more information on ``iterator`` and ``chunksize``.

compression : str or dict, default 'infer'
    For on-the-fly decompression of on-disk data. If 'infer' and 'filepath_or_buffer' is
    path-like, then detect compression from the following extensions: '.gz',
    '.bz2', '.zip', '.xz', '.zst', '.tar', '.tar.gz', '.tar.xz' or '.tar.bz2'
```

```
    (otherwise no compression).
    If using 'zip' or 'tar', the ZIP file must contain only one data file to be read in.
    Set to ``None`` for no decompression.
    Can also be a dict with key ``'method'`` set
    to one of {``'zip'``, ``'gzip'``, ``'bz2'``, ``'zstd'``, ``'xz'``, ``'tar'``} and
    other key-value pairs are forwarded to
    ``zipfile.ZipFile``, ``gzip.GzipFile``,
    ``bz2.BZ2File``, ``zstandard.ZstdDecompressor``, ``lzma.LZMAFile`` or
    ``tarfile.TarFile``, respectively.
    As an example, the following could be passed for Zstandard decompression using a
    custom compression dictionary:
    ``compression={'method': 'zstd', 'dict_data': my_compression_dict}``.

    .. versionadded:: 1.5.0
        Added support for `.tar` files.

    .. versionchanged:: 1.4.0 Zstandard support.

thousands : str (length 1), optional
    Character acting as the thousands separator in numerical values.
decimal : str (length 1), default '.'
    Character to recognize as decimal point (e.g., use ',' for European data).
lineterminator : str (length 1), optional
    Character used to denote a line break. Only valid with C parser.
quotechar : str (length 1), optional
    Character used to denote the start and end of a quoted item. Quoted
    items can include the ``delimiter`` and it will be ignored.
quoting : {0 or csv.QUOTE_MINIMAL, 1 or csv.QUOTE_ALL, 2 or csv.QUOTE_NONNUMERIC, 3 or csv.QUOTE_NONE}, default
csv.QUOTE_MINIMAL
    Control field quoting behavior per ``csv.QUOTE_*`` constants. Default is
    ``csv.QUOTE_MINIMAL`` (i.e., 0) which implies that only fields containing special
    characters are quoted (e.g., characters defined in ``quotechar``, ``delimiter``,
    or ``lineterminator``.
doublequote : bool, default True
   When ``quotechar`` is specified and ``quoting`` is not ``QUOTE_NONE``, indicate
   whether or not to interpret two consecutive ``quotechar`` elements INSIDE a
   field as a single ``quotechar`` element.
escapechar : str (length 1), optional
    Character used to escape other characters.
comment : str (length 1), optional
    Character indicating that the remainder of line should not be parsed.
    If found at the beginning
    of a line, the line will be ignored altogether. This parameter must be a
    single character. Like empty lines (as long as ``skip_blank_lines=True``),
    fully commented lines are ignored by the parameter ``header`` but not by
    ``skiprows``. For example, if ``comment='#'``, parsing
    ``#empty\na,b,c\n1,2,3`` with ``header=0`` will result in ``'a,b,c'`` being
    treated as the header.
encoding : str, optional, default 'utf-8'
    Encoding to use for UTF when reading/writing (ex. ``'utf-8'``). `List of Python
    standard encodings
    <https://docs.python.org/3/library/codecs.html#standard-encodings>`_ .

encoding_errors : str, optional, default 'strict'
    How encoding errors are treated. `List of possible values
    <https://docs.python.org/3/library/codecs.html#error-handlers>`_ .

    .. versionadded:: 1.3.0

dialect : str or csv.Dialect, optional
    If provided, this parameter will override values (default or not) for the
    following parameters: ``delimiter``, ``doublequote``, ``escapechar``,
    ``skipinitialspace``, ``quotechar``, and ``quoting``. If it is necessary to
    override values, a ``ParserWarning`` will be issued. See ``csv.Dialect``
    documentation for more details.
on_bad_lines : {'error', 'warn', 'skip'} or Callable, default 'error'
    Specifies what to do upon encountering a bad line (a line with too many fields).
    Allowed values are :

    - ``'error'``, raise an Exception when a bad line is encountered.
    - ``'warn'``, raise a warning when a bad line is encountered and skip that line.
    - ``'skip'``, skip bad lines without raising or warning when they are encountered.

    .. versionadded:: 1.3.0

    .. versionadded:: 1.4.0

        - Callable, function with signature
          ``(bad_line: list[str]) -> list[str] | None`` that will process a single
          bad line. ``bad_line`` is a list of strings split by the ``sep``.
          If the function returns ``None``, the bad line will be ignored.
          If the function returns a new ``list`` of strings with more elements than
          expected, a ``ParserWarning`` will be emitted while dropping extra elements.
```

```
        Only supported when ``engine='python'``

    .. versionchanged:: 2.2.0

        - Callable, function with signature
          as described in `pyarrow documentation
          <https://arrow.apache.org/docs/python/generated/pyarrow.csv.ParseOptions.html
          #pyarrow.csv.ParseOptions.invalid_row_handler>`_ when ``engine='pyarrow'``

delim_whitespace : bool, default False
    Specifies whether or not whitespace (e.g. ``' '`` or ``'\t'``) will be
    used as the ``sep`` delimiter. Equivalent to setting ``sep='\s+'``. If this option
    is set to ``True``, nothing should be passed in for the ``delimiter``
    parameter.

    .. deprecated:: 2.2.0
        Use ``sep="\s+"`` instead.
low_memory : bool, default True
    Internally process the file in chunks, resulting in lower memory use
    while parsing, but possibly mixed type inference.  To ensure no mixed
    types either set ``False``, or specify the type with the ``dtype`` parameter.
    Note that the entire file is read into a single :class:`~pandas.DataFrame`
    regardless, use the ``chunksize`` or ``iterator`` parameter to return the data in
    chunks. (Only valid with C parser).
memory_map : bool, default False
    If a filepath is provided for ``filepath_or_buffer``, map the file object
    directly onto memory and access the data directly from there. Using this
    option can improve performance because there is no longer any I/O overhead.
float_precision : {'high', 'legacy', 'round_trip'}, optional
    Specifies which converter the C engine should use for floating-point
    values. The options are ``None`` or ``'high'`` for the ordinary converter,
    ``'legacy'`` for the original lower precision pandas converter, and
    ``'round_trip'`` for the round-trip converter.

storage_options : dict, optional
    Extra options that make sense for a particular storage connection, e.g.
    host, port, username, password, etc. For HTTP(S) URLs the key-value pairs
    are forwarded to ``urllib.request.Request`` as header options. For other
    URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are
    forwarded to ``fsspec.open``. Please see ``fsspec`` and ``urllib`` for more
    details, and for more examples on storage options refer `here
    <https://pandas.pydata.org/docs/user_guide/io.html?
    highlight=storage_options#reading-writing-remote-files>`_.

dtype_backend : {'numpy_nullable', 'pyarrow'}, default 'numpy_nullable'
    Back-end data type applied to the resultant :class:`DataFrame`
    (still experimental). Behaviour is as follows:

    * ``"numpy_nullable"``: returns nullable-dtype-backed :class:`DataFrame`
      (default).
    * ``"pyarrow"``: returns pyarrow-backed nullable :class:`ArrowDtype`
      DataFrame.

    .. versionadded:: 2.0

Returns
-------
DataFrame or TextFileReader
    A comma-separated values (csv) file is returned as two-dimensional
    data structure with labeled axes.

See Also
--------
DataFrame.to_csv : Write DataFrame to a comma-separated values (csv) file.
read_table : Read general delimited file into DataFrame.
read_fwf : Read a table of fixed-width formatted lines into DataFrame.

Examples
--------
>>> pd.read_csv('data.csv')  # doctest: +SKIP
File:      c:\users\hp\anaconda3\lib\site-packages\pandas\io\parsers\readers.py
Type:      function
```

In [4]:
```
# printing the first 5 rows of the dataset
diabetes_dataset.head()
```

Out[4]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In [5]:

```
# number of rows and Columns in this dataset
diabetes_dataset.shape
```

Out[5]:

```
(768, 9)
```

In [6]:

```
# getting the statistical measures of the data
diabetes_dataset.describe()
```

Out[6]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	C
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	(
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	(
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	(
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	(
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	(
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	'
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	'

In [7]:

```
diabetes_dataset['Outcome'].value_counts()
```

Out[7]:

```
Outcome
0    500
1    268
Name: count, dtype: int64
```

0 --> Non-Diabetic

1 --> Diabetic

In [8]:

```
diabetes_dataset.groupby('Outcome').mean()
```

Out[8]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
Outcome								
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	0.429734	31.190000
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	0.550500	37.067164

In [9]:

```
# separating the data and labels
X = diabetes_dataset.drop(columns = 'Outcome', axis=1)
Y = diabetes_dataset['Outcome']
```

In [10]:

```
print(X)
```

```
     Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0              6      148             72             35        0  33.6   
1              1       85             66             29        0  26.6   
2              8      183             64              0        0  23.3   
3              1       89             66             23       94  28.1   
4              0      137             40             35      168  43.1   
..           ...      ...            ...            ...      ...   ...   
763           10      101             76             48      180  32.9   
764            2      122             70             27        0  36.8   
765            5      121             72             23      112  26.2   
766            1      126             60              0        0  30.1   
767            1       93             70             31        0  30.4   

     DiabetesPedigreeFunction  Age  
0                       0.627   50  
1                       0.351   31  
2                       0.672   32  
3                       0.167   21  
4                       2.288   33  
..                        ...  ...  
763                     0.171   63  
764                     0.340   27  
765                     0.245   30  
766                     0.349   47  
767                     0.315   23  

[768 rows x 8 columns]
```

In [11]:
```
print(Y)
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Data Standardization

In [12]:
```
scaler = StandardScaler()
```

In [13]:
```
scaler.fit(X)
```

Out[13]:
```
▾ StandardScaler
StandardScaler()
```

In [14]:
```
standardized_data = scaler.transform(X)
```

In [15]:
```
print(standardized_data)
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
 ...
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336
  -0.27575966]
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101
   1.17073215]
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505
  -0.87137393]]
```

In [16]:
```
X = standardized_data
Y = diabetes_dataset['Outcome']
```

In [17]:
```
print(X)
print(Y)
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
 ...
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336
  -0.27575966]
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101
   1.17073215]
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505
  -0.87137393]]
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Train Test Split

In [18]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2, stratify=Y, random_state=2)
```

In [19]:
```
print(X.shape, X_train.shape, X_test.shape)
```

```
(768, 8) (614, 8) (154, 8)
```

Training the Model

In [20]:
```
classifier = svm.SVC(kernel='linear')
```

In [21]:
```
#training the support vector Machine Classifier
classifier.fit(X_train, Y_train)
```

Out[21]:
```
SVC
SVC(kernel='linear')
```

Model Evaluation

Accuracy Score

In [22]:
```
# accuracy score on the training data
X_train_prediction = classifier.predict(X_train)
training_data_accuracy = accuracy_score(X_train_prediction, Y_train)
```

In [23]:
```
print('Accuracy score of the training data : ', training_data_accuracy)
```

```
Accuracy score of the training data :  0.7866449511400652
```

In [24]:
```
# accuracy score on the test data
X_test_prediction = classifier.predict(X_test)
test_data_accuracy = accuracy_score(X_test_prediction, Y_test)
```

In [25]:
```
print('Accuracy score of the test data : ', test_data_accuracy)
```

```
Accuracy score of the test data :  0.7727272727272727
```

Making a Predictive System

In [28]:
```
input_data = (5,166,72,19,175,25.8,0.587,51)

# changing the input_data to numpy array
input_data_as_numpy_array = np.asarray(input_data)

# reshape the array as we are predicting for one instance
input_data_reshaped = input_data_as_numpy_array.reshape(1,-1)

# standardize the input data
std_data = scaler.transform(input_data_reshaped)
print(std_data)

prediction = classifier.predict(std_data)
print(prediction)
```

```
if (prediction[0] == 0):
  print('The person is not diabetic')
else:
  print('The person is diabetic')
```

```
[[ 0.3429808   1.41167241  0.14964075 -0.09637905  0.82661621 -0.78595734
   0.34768723  1.51108316]]
[1]
The person is diabetic
```

In [27]:
```
import warnings
warnings.filterwarnings("ignore")
```

Creating Visualizations

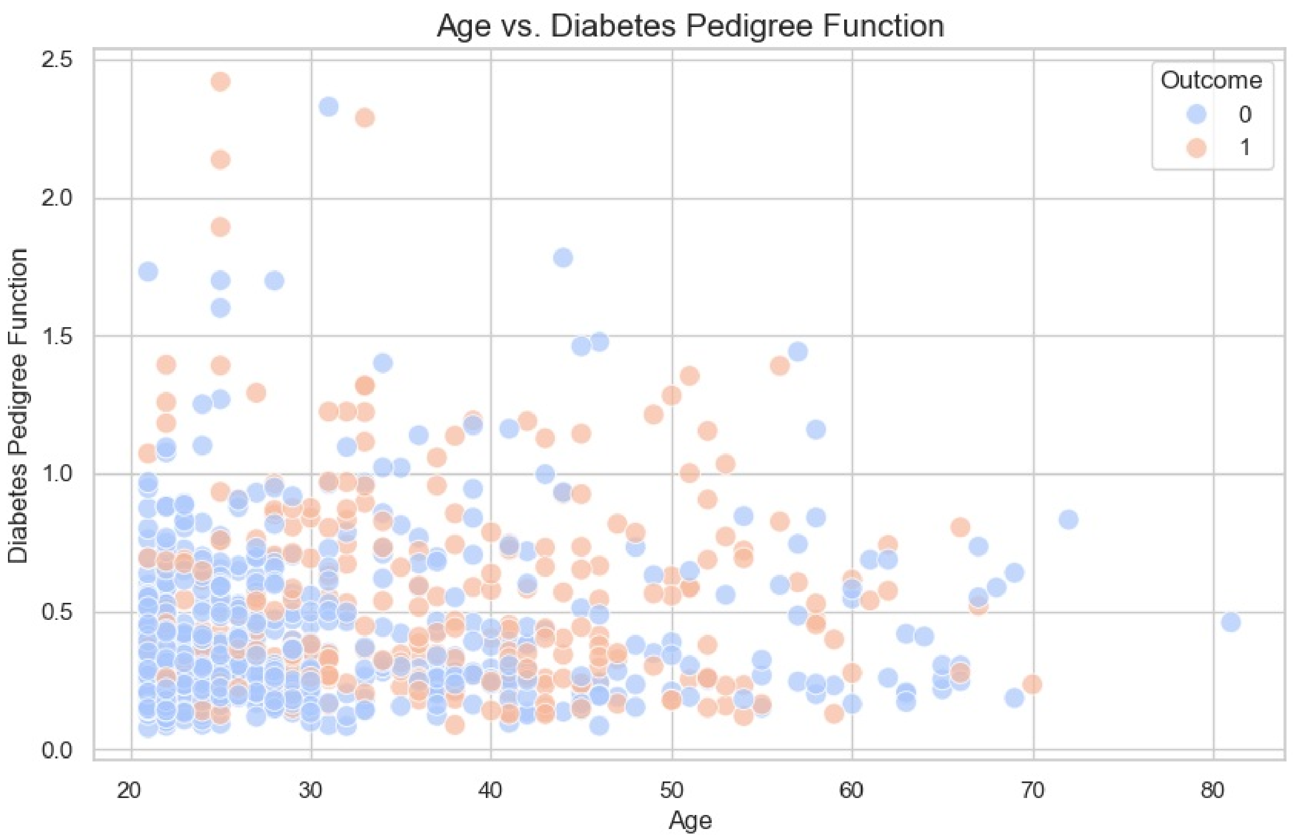
In [33]:
```
import matplotlib.pyplot as plt
import seaborn as sns
```

In [35]:
```
# Set the style for the plot
sns.set_style("whitegrid")

# Create the scatter plot
plt.figure(figsize=(10,6))  # Optional: Adjust the figure size
scatter_plot = sns.scatterplot(
    x='Age',
    y='DiabetesPedigreeFunction',
    hue='Outcome',
    palette='coolwarm',  # Customizes colors for 0 and 1
    data=diabetes_dataset,
    s=100,  # Size of the points
    alpha=0.7  # Transparency for better visualization
)

# Add titles and labels
plt.title('Age vs. Diabetes Pedigree Function', fontsize=15)
plt.xlabel('Age', fontsize=12)
plt.ylabel('Diabetes Pedigree Function', fontsize=12)

# Show the legend and plot
plt.legend(title='Outcome', loc='upper right')
plt.show()
```



INTEPRETATION The above plot showed that there are more non-diabetic patients for people lower than 30 years of age, however it was observed from the Diabetes Pedigree Function that a person is likely to be diabetic if there is a family history of diabetes. It was also observed from the plot that persons within 30 to 50 years of age are at a higher risk of having diabetes. Something very worthy of note from the plot is that it is very rare(though not impossible) for a geriatric who has a history of diabetes to be non-diabetic.

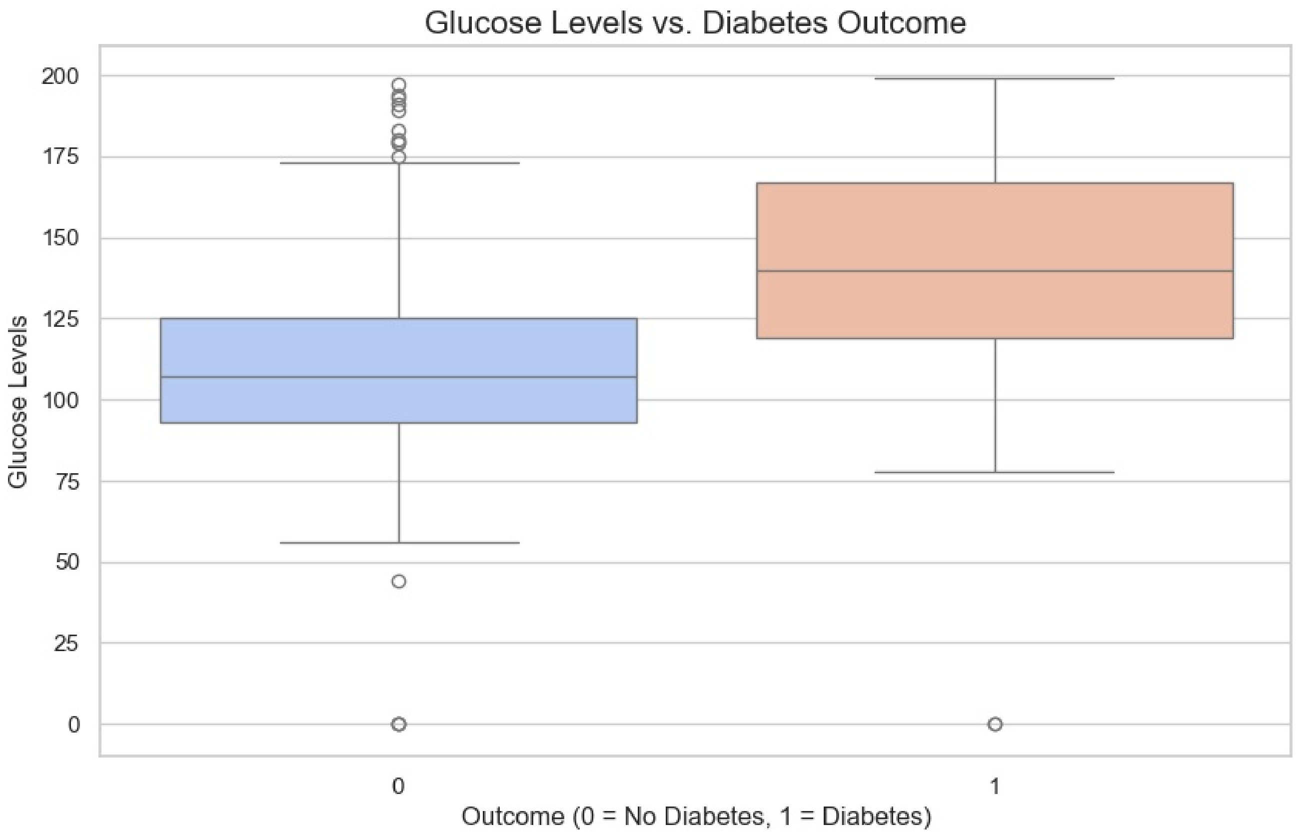
In [36]:

```
# Set the style for the plot
sns.set_style("whitegrid")

# Create the box plot
plt.figure(figsize=(10, 6))  # Adjust the figure size
box_plot = sns.boxplot(
    x='Outcome',     # Outcome on the x-axis (0 or 1)
    y='Glucose',     # Glucose levels on the y-axis
    data=diabetes_dataset,         # Your dataset
    palette='coolwarm'  # Color palette to differentiate between 0 and 1
)

# Add titles and labels
plt.title('Glucose Levels vs. Diabetes Outcome', fontsize=15)
plt.xlabel('Outcome (0 = No Diabetes, 1 = Diabetes)', fontsize=12)
plt.ylabel('Glucose Levels', fontsize=12)

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



INTEPRETATION The above box plot to used to check how glucose levels is correlated with the disease condition. It can be clearly seen that non-diabetic patients have a comparatively lower levels of glucose than the diabetic patients, so measures taken to reduce glucose levels could play a significant role in preventing diabetes.

CONCLUSION

In this study, we explored key features of the dataset related to diabetes prediction, focusing on the relationship between glucose levels, age, family history (Diabetes Pedigree Function), and the outcome of diabetes diagnosis. The visualizations provided insights into the distribution of glucose levels across diabetic and non-diabetic patients, highlighting a clear distinction in glucose levels between the two groups. These insights reinforce the importance of glucose levels and genetic factors in diabetes prediction, supporting the validity of the SVM model used in this study for accurate classification.