Appendix A. Code of the program

All code shown below has been written by Héctor García Cebollada as part of its work for this Master thesis. Some of the programs used along this code have some restrictions just for free use in research. Code is shown and distributed according to modules, as mentioned in the main text. Some of the parts of the code can be different for other versions of Python or Biopython.

PointMutation class (mutation.py)

```
from numpy import mean, std


class PointMutation(object):  # This defines a class for mutations of
a single aminoacid in proteins.
    ancestral = ""
    consensus = ""  # These are the things to evaluate in each
mutation.
    helix = []
    disulphide = ""
    acid_bonds = ""
    cavities = ""
    exposure = ""
    energy = ""
    measured = ""
    m_average = ""
    m_sd = ""
    proposed_by = ""

    score = ""  # This is the final score of the mutation. The higher,
the more probable the mutation is good

    def __init__(self, original, mutated, position, chain,
structure_id, start_number):
        self.original = original  # 1-letter code for the amino acid
in the original chain
        self.mutated = mutated  # 1-letter code for the amino acid in
the mutated chain
        self.position = int(position)  # Position of the mutation
        self.chain = chain  # Chain of the mutation
        self.structure_id = structure_id.lower()  # PDB code
        self.start_number = start_number
        self.seq_pos = self.position - int(start_number) + 1

    def __repr__(self):  # For proper printing
        return "%s%i%s chain %s PDB %s" % (self.original,
self.position, self.mutated, self.chain, self.structure_id.upper())

    def is_data_complete(self):  # Checks if all modules have
evaluated the mutation.
        data = [self. ancestral, self.consensus, self.helix,
self.disulphide, self.acid_bonds, self.cavities, self.exposure]  #
Modules
        incomplete = []
        i = 0
```

```
        for item in data:
            if item == "":
                incomplete.append(i)
            i += 1
        if incomplete != []:  # If some module hasn't evaluated yet
the mutation, it returns a code number for that module
            return incomplete
        else:
            return True  # If everything is evaluated, it goes on

    def complete_data(self, helix_list):  # This completes data if
incomplete.
        incomplete = self.is_data_complete()
        if incomplete != True:  # Here come the methods for getting
values for each module.
            if 0 in incomplete:
                self.ancestral = 0
            if 1 in incomplete:
                self.consensus = 0
            if 2 in incomplete:
                self = ["X", "X", "X"]
            if 3 in incomplete:
                self.disulphide = "disulphide"
            if 4 in incomplete:
                self.acid_bonds = "H bonds"
            if 5 in incomplete:
                self.cavities = "cavities"
            if 6 in incomplete:
                self.exposure = "exposure"

    def complete_measured(self):
        self.m_average = mean(self.measured)
        self.m_sd = std(self.measured)

    def write_full_mutation(self):
        return("%s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s" %
               (self.structure_id, self.chain, self.original,
str(self.position),
                self.mutated, str(self.ancestral),
str(self.consensus),
                self.helix[0], str(self.helix[1]), str(self.helix[2]),
                str(self.disulphide), str(self.acid_bonds),
str(self.exposure),
                str(self.energy), str(self.cavities),
str(self.m_average), str(self.m_sd)))
```

Retrieval (clean_retrieval.py)

```
from Bio import PDB, SeqIO, AlignIO
from Bio.PDB import PDBList
import os


def retrieve_PDB_file(structure_id):
    directory = "running_projects/%s" % structure_id
    pdbl = PDBList()
    pdbl.retrieve_pdb_file(structure_id, pdir=directory, file_format=
"pdb")  # Download PDB in pdir folder, named pdb[ID].ent
    dir_list = os.listdir(directory)
    if "PAML" not in dir_list:
        os.mkdir("running_projects/%s/PAML" % structure_id)
```

```
def read_structure(structure_id):
    p = PDB.PDBParser(PERMISSIVE=1)  # Opens the parser for using PDB
structures, trying to solve misannotations
    structure = p.get_structure(structure_id,
"running_projects/%s/pdb%s.ent" % (structure_id, structure_id))
    return structure


def read_sequence(structure_id):
    # Reading annotated sequence
    myseq_full = SeqIO.parse("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id), "pdb-seqres")
    chain_list_full = {}  # Dictionary for reducing multiplicity of
equal chains
    for chain in myseq_full:
        if chain.seq not in chain_list_full:
            chain_list_full[chain.seq] = [chain.id]  # Adds new chains
in a new entry
        else:
            chain_list_full[chain.seq].append(chain.id)
            # Adds repeated chains to the already existing ones,
adding the name of the new chains
    # Reading sequence appearing in PDB
    myseq_PDB = SeqIO.parse("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id), "pdb-atom")
    chain_list_PDB = {}  # Same than before but with PDB (incomplete)
sequence
    for chain in myseq_PDB:
        if chain.seq not in chain_list_PDB:
            chain_list_PDB[chain.seq] = [chain.id]
        else:
            chain_list_PDB[chain.seq].append(chain.id)
    return chain_list_full, chain_list_PDB
```

Sequence analysis (clean_consensus.py)

```
from Bio import SeqIO, AlignIO, pairwise2
from Bio.Align import AlignInfo
from Bio.Align.Applications import MuscleCommandline
from Bio.Blast import NCBIWWW, NCBIXML
from Bio.Phylo.Applications import PhymlCommandline
from Bio.Phylo.PAML import codeml
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.SubsMat.MatrixInfo import blosum62
from Bio.Application import ApplicationError
from io import StringIO
import subprocess
import os
from mutation import PointMutation


def read_sequence(structure_id):
    # Reading annotated sequence
    myseq_full = SeqIO.parse("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id), "pdb-seqres")
    chain_list_full = {}  # Dictionary for reducing multiplicity of
equal chains
    chain_codes = {}
```

```
    for chain in myseq_full:
        if chain.seq not in chain_list_full:
            ident = chain.id
            ident = ident.split(":")
            ident = ident[len(ident)-1]
            chain_list_full[chain.seq] = ident  # Adds new chains in a
new entry
            chain_codes[ident] = [ident]
        else:
            ident = chain.id
            ident = ident.split(":")
            ident = ident[len(ident) - 1]
            chain_codes[chain_list_full[chain.seq]].append(ident)
            # Adds repeated chains to the already existing ones,
adding the name of the new chains
    # Reading sequence appearing in PDB. Residues not appearing in PDB
are usually very flexible, difficult to trust.
    myseq_PDB = SeqIO.parse("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id), "pdb-atom")
    chain_list_PDB = {}  # Same than before but with PDB (incomplete)
sequence
    for chain in myseq_PDB:
        if chain.seq not in chain_list_PDB:
            ident = chain.id
            ident = ident.split(":")
            ident = ident[len(ident)-1]
            chain_list_PDB[chain.seq] = [ident]
        else:
            ident = chain.id
            ident = ident.split(":")
            ident = ident[len(ident) - 1]
            chain_list_PDB[chain.seq].append(ident)
    return chain_list_full, chain_codes, chain_list_PDB


def get_alignment(chain_list_full, structure_id):
    for chain in chain_list_full:
        handle = NCBIWWW.qblast("blastp", "nr", chain,
hitlist_size=10000, gapcosts="10 1", expect=0.0001)  # With
nonredundant database
        blast_record = NCBIXML.read(handle)  # (handle) is used as
argument without saving a BLAST file
        ids = ["my_protein"]  # The given sequence is the only one
without ID in BLAST, so this one is given
        sequences = [str(chain)]  # Further sequence clustering and
alignments need to be performed
        for alignment in blast_record.alignments:
            for hsp in alignment.hsps:
                if (hsp.identities/float(hsp.align_length)) > 0.7 and
"Z" not in hsp.sbjct and "B" not in hsp.sbjct:
                    sequences.append(hsp.sbjct)  # Takes the other
sequences (the aligned part) and gets it into a list
                    name = alignment.title
                    while len(name) > 10:
                        name = name[0:10]
                        if name in ids:
                            name = "A" + name
                    ids.append(alignment.title)  # Takes the title of
the other sequences
        short_ids = []  # Empty lists for the records' properties
        desc_list = []
```

```
        for item in ids:
            desc = "NA"  # For proteins without description, NA will
be theirs (only expected in "my protein")
            if "|" in item:
                index = 0
                for i in range(4):
                    index = item.find("|", index+1, len(item))  # For
getting the short id (just until the fourth bar)
                desc = item[index+1:]  # The rest after (+1) the
fourth bar is the description
                item = item[3:index]  # The three first characters
indicate the source, if not deleted, it may lead to
                                      # errors in writing strict
Phylip such as getting two identical IDs from two
                                      # different sequences.
            short_ids.append(item)
            desc_list.append(desc)
        fasta_list = []  # To keep all sequences in a list.
        for i in range(len(sequences)):
            my_seq = SeqRecord(Seq(sequences[i]))  # Make a SeqRecord
with each sequence
            my_seq.id = short_ids[i]  # Add their properties to each
SeqRecord (they are the same order in the list)
            my_seq.description = desc_list[i]
            fasta_list.append(my_seq)  # Append SeqRecord objects to a
new list, for writing them
        doc_handle =
"/home/elhectro2/Documentos/TFM/Programming_TFM/running_projects/%s/%s
_in.faa" \
                     % (structure_id, chain_list_full[chain])
        align_handle =
"/home/elhectro2/Documentos/TFM/Programming_TFM/running_projects/%s/%s
_align.faa" \
                     % (structure_id, chain_list_full[chain])
        SeqIO.write(fasta_list, doc_handle, "fasta")  # Writes the
SeqRecord objects as sequences in the file doc_handle
        if len(fasta_list) >= 250:
            SeqIO.write(fasta_list[0:250], align_handle, "fasta")
        else:
            SeqIO.write(fasta_list[0:len(fasta_list)], align_handle,
"fasta")
        threshold_cdhit = "0.9"
        if len(chain) < 100:
            threshold_cdhit = "0.8"
        out_cdhit =
"/home/elhectro2/Documentos/TFM/Programming_TFM/running_projects/%s/%s
_cdhit" \
                     % (structure_id, chain_list_full[chain])
        subprocess.call(["/usr/bin/cdhit-master/cd-hit", "-i",
doc_handle, "-o", out_cdhit, "-c", threshold_cdhit])
        file_out = open(out_cdhit, "r")
        not_protein = True
        for line in file_out.readlines():
            if "my_protein" in line:
                not_protein = False
                break
        file_out.close()
        file_out = open(out_cdhit, "r")
        if not_protein:
            out_cdhit =
"/home/elhectro2/Documentos/TFM/Programming_TFM/running_projects/%s/%s
```

```
_cdhit_OK" \
                    % (structure_id, chain_list_full[chain])
            file = open(out_cdhit, "w")
            for line in file_out:
                file.write(line)
            file_out.close()
            file_in = open(doc_handle)
            line = file_in.readline()
            while "my_protein" not in line:
                line = file_in.readline()
            else:
                file.write(line)
                line = file_in.readline()
                while ">" not in line:
                    file.write(line)
                    line = file_in.readline()
            file.close()
        # The line before is $/usr/bin/cdhit-master/cd-hit -i
doc_handle -o out_cdhit, with the definitions set before.
        # Default settings are used for CD-hit (90% identity in
clustering)
        muscle_cline = MuscleCommandline(input=out_cdhit)  # Command
for using MUSCLE in the following line.
                                                           # If not
used, not working
        stdout, stderr = muscle_cline()
        alignment = AlignIO.read(StringIO(stdout), "fasta")  # Allows
to use the result without writing it
        PAML_handle = open("running_projects/%s/%s_alignment.phy" %
(structure_id, chain_list_full[chain]), "w")
        AlignIO.write(alignment, PAML_handle, "phylip-sequential")
        backup =
"/home/elhectro2/Documentos/TFM/Programming_TFM/running_projects/%s/%s
_in.faa" \
                 % (structure_id, chain_list_full[chain])
        muscle_cline = MuscleCommandline(input=backup)
        stdout, stderr = muscle_cline()
        alignment = AlignIO.read(StringIO(stdout), "fasta")  # Allows
to use the result without writing it
        PAML_handle =
open("running_projects/%s/%s_alignment_backup.phy" % (structure_id,
chain_list_full[chain]), "w")
        try:
            AlignIO.write(alignment, PAML_handle, "phylip-sequential")
        except ValueError:
            pass


def consensus_alignment(structure_id):
    chain = "A"
    if structure_id in "2ci21acb":
        chain = "I"
    file = "running_projects/%s/%s_align.faa" % (structure_id, chain)
    muscle cline = MuscleCommandline(input=file)
    stdout, stderr = muscle_cline()
    alignment = AlignIO.read(StringIO(stdout), "fasta")
    return alignment


def evaluate_consensus_mutation(alignment, mutation):
    f_position = mutation.seq_pos
```

```
    reference_seq = "NA"
    index = 0
    position = 1
    for record in alignment:
        if "my_protein" in record.id:
            reference_seq = record.seq
    while position <= f_position:
        if reference_seq[index] != "-":
            position += 1
        index += 1
    index -= 1
    total = 0
    wt = 0
    mut = 0
    for record in alignment:
        letter = record.seq[index]
        if letter != "-":
            total += 1
        if letter == mutation.mutated:
            mut += 1
        if letter == mutation.original:
            wt += 1
    score = (mut-wt)/float(total)
    return score


def evaluate_dumb_consensus(chain_list_full, structure_id,
start_number):  # Obsolete. Kept in case something is useful
    consensus_mut = []
    consensus_seq = []
    for chain in chain_list_full:
        file = "running_projects/%s/%s_align.faa" % (structure_id,
chain_list_full[chain])
        muscle_cline = MuscleCommandline(input=file)
        stdout, stderr = muscle_cline()
        alignment = AlignIO.read(StringIO(stdout), "fasta")
        summary_align = AlignInfo.SummaryInfo(alignment)  # Object for
studying properties of the alignment
        consensus = summary_align.dumb_consensus(threshold=0.3)  #
Makes the simple consensus, with X as the no-consensus symbol
        consensus_seq.append([chain_list_full[chain], str(consensus)])
        res = pairwise2.align.globalds(chain, consensus, blosum62, -
10, -0.5)
        seq1 = res[0][0]
        seq2 = res[0][1]
        number = 0
        for i in range(len(seq1)):
            if seq1[i] != "-":
                number += 1
                if seq1[i] != seq2[i] and seq2[i] not in "X-":
                    new_mutation = PointMutation(seq1[i], seq2[i],
number, chain_list_full[chain], structure_id, start_number)
                    new_mutation.consensus =
evaluate consensus mutation(alignment, new mutation)
                    consensus_mut.append(new_mutation)
        # Compares consensus with original sequence character by
character. If a different character is found
        # between both(other than the no-consensus symbol), that
mutation is suggested as a result
    return consensus_seq, consensus_mut
```

```
def clean_alignment(chain_list_full, structure_id):  # Prepares the
alignment file for later steps
    for chain in chain_list_full:
        file_in = open("running_projects/%s/%s_alignment.phy" %
(structure_id, chain_list_full[chain]))
        file_out = open("running_projects/%s/%s_alignment_OK.phy" %
(structure_id, chain_list_full[chain]), "w")
        first = file_in.readline()  # First line are two numbers, with
the number of sequences and its length
        file_out.write(first)
        seqnum, length = first.split()
        for i in range(0, int(seqnum)):  # Reads as many lines as
sequences there are.
            line = file_in.readline()
            name = line[0:10]  # Names in strict phylip are always 10
characters long.
            new_name = ""  # String for the corrected ID
            for char in name:
                if char == "|":  # | symbol has some meaning in trees,
it should be avoided.
                    new_name += "I"
                else:
                    new_name += char
            seq = line[10:len(line)]
            file_out.write("%s  %s" % (new_name, seq))  # A double
space is added between ID and sequence
                                                        # in order for
PAML and PhyML to work without errors.
        for line in file_in.readlines(): # This writes any line out of
the alignment.
            file_out.write(line)
        file_in.close()
        file_out.close()
        try:
            file_in =
open("running_projects/%s/%s_alignment_backup.phy" % (structure_id,
chain_list_full[chain]))
            file_out =
open("running_projects/%s/%s_alignment_backup_OK.phy" % (structure_id,
chain_list_full[chain]), "w")
            first = file_in.readline()  # First line are two numbers,
with the number of sequences and its length
            file_out.write(first)
            seqnum, length = first.split()
            for i in range(0, int(seqnum)):  # Reads as many lines as
sequences there are.
                line = file_in.readline()
                name = line[0:10]  # Names in strict phylip are always
10 characters long.
                new_name = ""  # String for the corrected ID
                for char in name:
                    if char == "|":  # | symbol has some meaning in
trees, it should be avoided.
                        new_name += "I"
                    else:
                        new_name += char
                seq = line[10:len(line)]
                file_out.write("%s  %s" % (new_name, seq))  # A double
space is added between ID and sequence
                # in order for PAML and PhyML to work without errors.
```

```
            for line in file_in.readlines():  # This writes any line
out of the alignment.
                file_out.write(line)
            file in.close()
            file_out.close()
        except ValueError:
            pass


def fast_tree(chain_list_full, structure_id):  # PhyML creates ML
trees in a fast way.
    mode = []
    for chain in chain_list_full:
        try:
            filename = "running_projects/%s/%s_alignment_OK.phy" %
(structure_id, chain_list_full[chain])
            cmd = PhymlCommandline(sequential=True, input=filename,
datatype="aa", model="JTT")  # Prepares use of PhyML
            out_log, err_log = cmd()  # Uses PhyML with the parameters
before.
            mode.append("OK")
        except ApplicationError:
            filename =
"running_projects/%s/%s_alignment_backup_OK.phy" % (structure_id,
chain_list_full[chain])
            cmd = PhymlCommandline(sequential=True, input=filename,
datatype="aa", model="JTT")  # Prepares use of PhyML
            out_log, err_log = cmd()  # Uses PhyML with the parameters
before.

os.rename("running_projects/%s/%s_alignment_backup_OK.phy_phyml_tree.t
xt" % (structure_id, chain_list_full[chain]),

"running_projects/%s/%s_alignment_OK.phy_phyml_tree.txt" %
(structure_id, chain_list_full[chain]))
            mode.append("no")
    return mode


def clean_tree(chain_list_full, structure_id, mode):  # Prepares the
tree file from PhyML to PAML.
    result = []
    j = 0
    for chain in chain_list_full:
        try:
            chain_mode = mode[j]
        except IndexError:
            chain_mode = "ok"
        j += 1
        in_file = open(
            "running_projects/%s/%s_alignment_OK.phy_phyml_tree.txt" %
(structure_id, chain_list_full[chain]))
        if chain_mode == "no":
            seq file =
open("running_projects/%s/%s_alignment_backup_OK.phy" % (structure_id,
chain_list_full[chain]))
        else:
            seq_file = open("running_projects/%s/%s_alignment_OK.phy"
% (structure id, chain list full[chain]))
        out_file = open(
            "running_projects/%s/%s_alignment_OK1.phy_phyml_tree.txt"
```

```
% (structure_id, chain_list_full[chain]),
            "w")
        number = seq_file.readline().split()[0]  # Gets the number of
sequences from the alignment's first line.
        seq_file.close()
        out_file.write("%s 1\n" % number)  # In the first line, there
must be the number of sequences and trees (1)
        for line in in_file.readlines():  # Usually it's only one
line, but this is used just in case
            i = 0  # Counter for the length of the line
            new_line = ""
            while i < len(line):
                if line[i] != ")":  # There are some numbers after )
that should be deleted
                    new_line += line[i]
                    i += 1
                else:
                    new_line += line[i]
                    i += 1
                    while line[i] not in ":;)" and i < len(line):  #
That numbers end in : ; or ) signs
                        i += 1
            out_file.write(new_line)
        out_file.close()
        in_file.close()
        result.append([chain_list_full[chain],number])
    return result


def PAML_tree(chain_list_full, structure_id, mode):  # PAML is
executed for getting the ancestral sequence
    i = 0
    for chain in chain_list_full:
        chain_mode = mode[i]
        i += 1
        if chain_mode == "no":
            filename =
"running_projects/%s/%s_alignment_backup_OK.phy" % (structure_id,
chain_list_full[chain])
        else:
            filename = "running_projects/%s/%s_alignment_OK.phy" %
(structure_id, chain_list_full[chain])
        treename =
"running_projects/%s/%s_alignment_OK1.phy_phyml_tree.txt" %
(structure_id, chain_list_full[chain])
        out = "running_projects/%s/PAML%s.out" % (structure_id,
chain_list_full[chain])
        work = "running_projects/%s/PAML/%s" % (structure_id,
chain_list_full[chain])
        cml = codeml.Codeml(alignment=filename, tree=treename,
out_file=out, working_dir=work)
        cml.set_options(noisy=1, verbose=0, runmode=0, seqtype=2,
CodonFreq=2, ndata=1, clock=1, aaDist=1, model=2,
                        aaRatefile="/bin/paml4.9e/dat/jones.dat",
NSsites=[0], icode=0, Mgene=0, fix_kappa=0, kappa=2,
                        fix_omega=0, omega=.4, fix_alpha=1, alpha=0,
Malpha=0, ncatG=None, getSE=0, RateAncestor=1,
                        Small_Diff=.5e-6, cleandata=0, fix_blength=1,
method=0, rho=1, fix rho=0)
        cml.run(command="/bin/paml4.9e/bin/codeml", parse=False,
verbose=False)
```

```
def read_ancestral_seq(chain_list_full, structure_id):  # The
ancestral sequence and its probabilities are in rst file.
    results_dict = {}
    for chain in chain_list_full:
        file = open("running_projects/%s/PAML/%s/rst" % (structure_id,
chain_list_full[chain]))
        my_dict = {}
        while True:
            line = file.readline()
            if "Prob of best state at each node, listed by site" in
line:  # This line shows the beginning of the results
                break
        for i in range(3):  # There are 3 lines without useful
information
            file.readline()
        for line in file.readlines():
            try:
                line = line.split()  # Result of this is [site, freq,
extant residues, predicted residue 1, 2, n]
                code = line[3]  # The first predicted residue is the
common ancestor's sequence
                letter = code[0]  # Structure is "one-letter code
(probability)", so first letter is the residue
                prob = float(code[2:len(code)-1])  # Then, the rest
without the parenthesis is the probability
                my_dict[int(line[0])] = [letter, prob]  # A
dictionary: for each position, its residue and probability
            except IndexError:
                break
        results_dict[chain_list_full[chain]] = my_dict  # The
predicted ancestral sequence is added for each chain
    return results_dict


def get_ancestral_seq(chain_list_full, structure_id):
    result = []
    for chain in chain_list_full:
        file = open("running_projects/%s/PAML/%s/rst" % (structure_id,
chain_list_full[chain]))
        sequence = ""
        values = []
        while True:
            line = file.readline()
            if "Prob of best state at each node, listed by site" in
line:  # This line shows the beginning of the results
                break
        for i in range(3):  # There are 3 lines without useful
information
            file.readline()
        for line in file.readlines():  # Then, the lines with useful
info are read
            try:
                line = line.split()  # Result of this is [site, freq,
extant residues, predicted residue 1, 2, n]
                code = line[3]  # The first predicted residue is the
common ancestor's sequence
                letter = code[0]  # Structure is "one-letter code
(probability)", so first letter is the residue
                value = float(code[2:len(code)-1])
```

```
                values.append(value)
                sequence += letter
            except IndexError:
                file.close()
                break
        file = open("running_projects/%s/PAML/%s/rst" % (structure_id,
chain_list_full[chain]))
        my_protein = ""
        for line in file.readlines():
            if "my_protein" in line and "(" not in line:
                my_protein = line.split()[1:len(line.split())]
                my_protein = "".join(my_protein)
                break
        result.append([chain_list_full[chain], sequence, my_protein,
values])
    return result


def compare(chain_list_full, structure_id, start_number):  # Uses the
previous function to compare the ancestral and protein sequence
    ancestral = read_ancestral_seq(chain_list_full, structure_id)
    result = []
    sequence = get_ancestral_seq(chain_list_full, structure_id)
    for chain in chain_list_full:
        for group in sequence:
            if group[0] == chain_list_full[chain]:
                r_chain = group[2]
        ancestral_chain = ancestral[chain_list_full[chain]]
        number = 0
        for i in range(len(r_chain)):
            if r_chain[i] != "-":
                number += start_number
                if r_chain[i] != ancestral_chain[i+1][0]:  # If a
difference between ancestral and extant exists, it proposes a
mutation.
                    new_mut = PointMutation(str(r_chain[i]),
str(ancestral_chain[i+1][0]), number, chain_list_full[chain],
structure_id, start_number)
                    new_mut.ancestral = ancestral_chain[i+1][1]  #
This is puntuated with its probability
                    result.append(new_mut)
    return result


def ancestral_and_consensus(structure_id, start_number):
    chain_list_full, chain_codes, chain_list_PDB =
read_sequence(structure_id)
    get_alignment(chain_list_full, structure_id)
    clean_alignment(chain_list_full, structure_id)
    mode = fast_tree(chain_list_full, structure_id)
    clean_tree(chain_list_full, structure_id, mode)
    PAML_tree(chain_list_full, structure_id, mode)
    ancestral_mutation = compare(chain_list_full, structure_id,
start number)
    ancestral = get_ancestral_seq(chain_list_full, structure_id)
    consensus, consensus_mutation =
evaluate_dumb_consensus(chain_list_full, structure_id, start_number)
    alignment = consensus_alignment(structure_id)
    return chain_list_PDB, ancestral, ancestral_mutation, alignment,
consensus_mutation
```

```
def get_data_mutation_ancestral(structure_id):
    chain_list_full, chain_codes, chain_list_PDB =
read sequence(structure id)
    get_alignment(chain_list_full, structure_id)
    clean_alignment(chain_list_full, structure_id)
    mode = fast_tree(chain_list_full, structure_id)
    cdhit_count = clean_tree(chain_list_full, structure_id, mode)
    PAML_tree(chain_list_full, structure_id, mode)
    ancestral = get_ancestral_seq(chain_list_full, structure_id)
    return cdhit_count, ancestral, chain_list_PDB


def get_calculated_ancestral(structure_id):
    chain_list_full, chain_codes, chain_list_PDB =
read_sequence(structure_id)
    ancestral = get_ancestral_seq(chain_list_full, structure_id)
    return ancestral


def evaluate_ancestral_mutation(ancestral, mutation):
    value = 0
    for item in ancestral:
        if item[0] == mutation.chain:
            or_sequence = item[2]
            anc_sequence = item[1]
            values = item[3]
            break
    pos = 0
    index = 0
    while int(pos) < int(mutation.seq_pos):
        letter = or_sequence[index]
        index += 1
        if letter != "-":
            pos += 1
    index -= 1
    anc = anc_sequence[index]
    if mutation.mutated == anc:
        value = values[index]
    return value
```

Alpha-helices (clean_alpha_helix.py)

```
from Bio.PDB.DSSP import dssp_dict_from_pdb_file
import subprocess
from mutation import PointMutation

exposure_dict = {"A": 73.2*1.05, "R": 178.9*1.05, "N": 109.2*1.05,
"D": 102.2*1.05, "C": 88.7*1.05, "E": 126.0*1.05,
                 "Q": 125.9*1.05, "G": 54.3*1.05, "H": 129.5*1.05,
"I": 122.5*1.05, "L": 131.9*1.05, "K": 149.9*1.05,
                 "M": 134.3*1.05, "F": 146.1*1.05, "P": 100.3*1.05,
"S": 76.0*1.05, "T": 93.3*1.05, "W": 173.2*1.05,
                 "Y": 156.9*1.05, "V": 102.2*1.05, "X": 118.7*1.05}
n_cap_dict = {"A": 1.1, "G": 0.35, "S": 0.25, "T": 0.48, "N": 0.0,
"D": 0.0, "Q": 1.25, "E": 0.8, "H": 1.18, "K": 1.1,
              "R": 1.1, "F": 1.18, "Y": 1.18, "W": 1.18, "L": 1.18,
"V": 1.18, "I": 1.18, "M": 1.18, "C": 1.18, "P": 1.54}
c_cap_dict = {"A": 0.4, "G": 0.0, "S": 0.32, "T": 0.32, "N": 0.0, "D":
0.76, "Q": 0.32, "E": 0.32, "H": 0.0, "K": 0.3,
              "R": 0.15, "F": 0.48, "Y": 0.48, "W": 0.48, "L": 0.48,
```

```
"V": 0.48, "I": 0.48, "M": 0.48, "C": 0.48, "P": 0.48}
inner_dict = {"A": 0.0, "G": 0.82, "S": 0.37, "T": 0.47, "N": 0.44,
"D": 0.40, "Q": 0.21, "E": 0.22, "H": 0.54, "K": 0.2,
              "R": 0.09, "F": 0.51, "Y": 0.53, "W": 0.53, "L": 0.19,
"V": 0.6, "I": 0.42, "M": 0.31, "C": 0.67, "P": 3.47}


def get_DSSP_result(structure_id):
    DSSP_result, keys =
dssp_dict_from_pdb_file("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id))
    return DSSP_result, keys


def get_relative_exposure(structure_id):
    DSSP_result, keys =
dssp_dict_from_pdb_file("running_projects/%s/pdb%s.ent" %
(structure_id, structure_id))
    result = []
    for key in keys:
        chain = key[0]
        aa = DSSP_result[key][0]
        if aa in "asdfghjklñqwertyuiopzxcvbnm":
            aa = "C"
        position = key[1][1]
        exp = DSSP_result[key][2] / exposure_dict[aa]
        result.append([chain, aa, position, exp])
    return result


def get_helices(DSSP_result, keys):
    helix_list = []
    previous_chain = "Ñ"
    # As PDB and DSSP work in English, Ñ will never be the name of a
chain and it will define blank previous data
    for key in keys:  # Iterates over the aminoacids in chain order
and in numerical order.
        if key[0] != previous_chain:  # When a new chain starts, the
one before is not related to their helices
            previous_chain = key[0]
            previous_aa = ""  # Because of that, blank previous
parameters are defined.
            previous_ss = ""  # These parameters are useful for
getting the N-cap and C-cap of the alpha helix.
            previous_key = ""  # Mostly for naming the N-cap and C-cap
            previous_exposure = ""
        aa = DSSP_result[key][0]
        ss = DSSP_result[key][1]
        if aa in "asdfghjklqwertyuiopzxcvbnm":
            aa = "C"
        exposure = DSSP_result[key][2] / exposure_dict[aa]
        if DSSP_result[key][1] == "H":
            # This means the aminoacid has the appropriate angles for
being in an alpha helix
            if previous_ss == "H":  # If the previous one is also H
structure, it cannot be the N-cap.
                helix.append([key[0], aa, key[1][1], exposure])
                # Thus, only the new aa is added to the helix
            else:  # If the previous one is not H structure, the
previous aa is the N-cap.
                if previous_aa != "":  # Unless the current aa is the
```

```
first in its chain.
                    helix = [[previous_chain, previous_aa,
previous_key[1][1], previous_exposure]]
                    # Assigning N-cap to helix
                    helix.append([key[0], aa, key[1][1], exposure])  #
Adding new aa to helix
                else:  # If the helix starts in the N-terminus
                    helix = [[key[0], aa, key[1][1], exposure]]
        elif previous_ss == "H":  # If the aa is not H structure, but
the one before is, it is the C-cap
            helix.append([key[0], aa, key[1][1], exposure])
            # C-cap is added at the end of the helix
            # Helix length is checked (It must be longer than 4
residues including N- and C-cap to be a turn of helix)
            if len(helix) >= 5:
                helix_list.append(helix)  # If the length is OK, the
helix is appended to the list of helices for improvement
        # The previous parameters are changed for the current ones to
move on to the next aa
        previous_aa = aa
        previous_ss = ss
        previous_key = key
        previous_exposure = exposure
    return helix_list


def HBPlus(structure_id):  # Calls HBPlus program to work. Creates
output file in the project path.
    input_dir = "running_projects/%s/pdb%s.ent" % (structure_id,
structure_id)
    subprocess.call(["/bin/hbplus/hbplus", input_dir])


def create_code(number, chain): # Creates an HBPlus output-like code
from the position and chain of a residue
    number = str(number)
    while len(number) < 4:
        number = "0" + number
    code = chain + number
    return code


def helix_full_code(helix_list):  # Creates a list of HBPlus output-
like codes for the residues in helices
    result = []
    for helix in helix_list:
        for residue in helix:  # Takes all residues of all helices to
check their position and chain
            number = residue[2]
            chain = residue[0]
            result.append(create_code(number, chain))  # Adds the
created codes to a list.
    return result


def relevant_acid_hbonds(structure_id):
    file = open("pdb%s.hb2" % structure_id, "r")
    relevant = []
    for line in file.readlines():
        if ("ASP" in line or "ASN" in line or "GLU" in line or "GLN"
in line) and "HOH" not in line:
```

```
            line = line.split()
            donor = line[0]
            acceptor = line[2]
            codes = {"ASP": "D", "GLU": "E", "GLN": "Q", "ASN": "N"}
            if ("ASP" in donor or "ASN" in donor or "GLU" in donor or
"GLN" in donor) and line[5][0] != "M":
                donor = [donor[0], str(int(donor[1:5])),
codes[donor[6:9]]]
                if donor not in relevant:
                    relevant.append(donor)
            if ("ASP" in acceptor or "ASN" in acceptor or "GLU" in
acceptor or "GLN" in acceptor) and line[5][1] != "M":
                acceptor = [acceptor[0], str(int(acceptor[1:5])),
codes[acceptor[6:9]]]
                if acceptor not in relevant:
                    relevant.append(acceptor)
    return relevant


def relevant_helix_hbonds(helix_list, structure_id):  # Filters the
results of HBPlus to get only helix relevant bonds.
    codes = helix_full_code(helix_list)  # Gets the codes for all
helices
    file = open("pdb%s.hb2" % structure_id, "r")  # Opens the HBPlus
output file
    relevant = []  # For storing results
    very_relevant = []
    for line in file.readlines():  # Each line is a H-bond
        lin = line.split()
        donor = lin[0][0:5]  # Uses only the part of the code
corresponding to the generated codes
        acceptor = lin[2][0:5]
        # Filters out bonds with water and bonds not concerning
residues in helices
        if (donor in codes or acceptor in codes) and "HOH" not in
lin[0] and "HOH" not in lin[2]:
            if "MM" not in lin[5]:  # Filters out backbone bonds
separated by 3 or 4 aa. Revise.
                if lin[5][0] != "M":
                    relevant.append(donor)  # Appends donors and
acceptors happening because of the side chain.
                if lin[5][1] != "M":
                    relevant.append(acceptor)
                if lin[5] == "SS":
                    very_relevant.append(donor)
                    very_relevant.append(acceptor)
    return relevant, very_relevant


def complete_helix_data(helix_list, structure_id):  # Adds H-bonds
information to the residue information in helix_list
    new_helix_list = []
    relevant, very_relevant = relevant_helix_hbonds(helix_list,
structure id)  # Gets the filtered results from the function before.
    for helix in helix_list:
        new_helix = []  # The same levels as the list before must be
kept.
        N_cap = helix[0]
        C_cap = helix[len(helix)-1]
        inner = helix[1:len(helix)-1]
        code_N_cap = create_code(N_cap[2], N_cap[0])
```

```
        code_C_cap = create_code(C_cap[2], C_cap[0])
        if code_N_cap in very_relevant:
            N_cap.append(False)
        else:
            N_cap.append(True)
        new_helix.append(N_cap)
        for residue in inner:
            number = residue[2]
            chain = residue[0]
            code = create_code(number, chain)  # To check if the
residue is involved or not in relevant H-bonds.
            if code in relevant:
                residue.append(False)  # If it is involved, it's not
recommendable to modify it.
            else:
                residue.append(True)  # If it is not, it can be
modified.
            new_helix.append(residue)
        if code_C_cap in very_relevant:
            C_cap.append(False)
        else:
            C_cap.append(True)
        new_helix.append(C_cap)
        new_helix_list.append(new_helix)
    return new_helix_list  # Returns the full list with all info


def propose_helix_mutations(helix_list, structure_id, start_number):
# With full information in helix_list, it proposes mutations.
    mutations = []
    # N-cap, C-cap and inner residues follow different selection rules
for aminoacid type.
    # Because of that, they are separated and evaluated differently.
    # Also, helix property is a 3 elements list: letter indicating N,
C-cap or inner; number evaluating improvement and
    # True or False for presence or absence of relevant H-bonds.
    for helix in helix_list:
        n_cap = helix[0]  # Helices prepared so that N-cap is the
first aa and C-cap the last
        c_cap = helix[len(helix)-1]
        inner = helix[1:len(helix)-1]
        if n_cap[3] > .10 and n_cap[1] not in "TDSGN" and n_cap[4]:  #
Test exposure, type of aminoacid and lack of HBonds
            new_mutation = PointMutation(n_cap[1], "D", n_cap[2],
n_cap[0], structure_id, start_number)
            new_mutation.helix = ["N", n_cap_dict[n_cap[1]], False]
            mutations.append(new_mutation)
        if c_cap[3] > .10 and c_cap[1] not in "GHN" and c_cap[4]:
            new_mutation = PointMutation(c_cap[1], "G", c_cap[2],
c_cap[0], structure_id, start_number)
            new_mutation.helix = ["C", c_cap_dict[c_cap[1]], False]
            mutations.append(new_mutation)
        for residue in inner:
            if residue[3] > .10 and residue[1] not in "ALRMK" and
residue[4]:
                new_mutation = PointMutation(residue[1], "A",
residue[2], residue[0], structure_id, start_number)
                new_mutation.helix = ["I", inner_dict[residue[1]],
False]
                mutations.append(new_mutation)
    return mutations
```

```
def main_helix_mutation(structure_id, start_number):
    DSSP result, keys = get DSSP result(structure id)
    helix_list = get_helices(DSSP_result, keys)
    HBPlus(structure_id)
    helix_list = complete_helix_data(helix_list, structure_id)
    mutations = propose_helix_mutations(helix_list, structure_id,
start_number)
    return helix_list, mutations


def evaluate_helix_mutation(helix_list, mutation):
    value = []
    for helix in helix_list:
        i = 0
        for residue in helix:
            if mutation.position == residue[2] and mutation.chain ==
residue[0] and mutation.original == residue[1]:
                if residue[4]:
                    h_bond = False
                else:
                    h_bond = True
                if i == 0:
                    value = ["N", n_cap_dict[mutation.original] -
n_cap_dict[mutation.mutated], h_bond]
                elif i == len(helix)-1:
                    value = ["C", c_cap_dict[mutation.original] -
c_cap_dict[mutation.mutated], h_bond]
                else:
                    value = ["I", inner_dict[mutation.original] -
inner_dict[mutation.mutated], h_bond]
            i += 1
    if value == []:
        value = ["X", "X", "X"]
    return value


def get_data_mutation_helix(structure_id):
    DSSP_result, keys = get_DSSP_result(structure_id)
    helix_list = get_helices(DSSP_result, keys)
    HBPlus(structure_id)
    helix_list = complete_helix_data(helix_list, structure_id)
    return helix_list
```

Disulphide bonds (clean_disulphide.py)

```
from mutation import PointMutation

three_to_one = {"CYS": "C", "ASP": "D", "SER": "S", "GLN": "Q", "LYS":
"K",
                "ILE": "I", "PRO": "P", "THR": "T", "PHE": "F", "ASN":
"N",
                "GLY": "G", "HIS": "H", "LEU": "L", "ARG": "R", "TRP":
"W",
                "ALA": "A", "VAL": "V", "GLU": "E", "TYR": "Y", "MET":
"M"}


def read_dbd_file(structure_id, ext):
    file = "DbD/%s.%s" % (structure_id.upper(), ext)
```

```
    file = open(file)
    result = []
    if ext == "csv":
        char = ","
        header = 10
    elif ext == "txt":
        char = None
        header = 20
    i = 0
    for line in file:
        i += 1
        if i > header:
            line = line.split(char)
            if len(line) > 2:
                res = line[0:6]
                res.append(line[7])
                result.append(res)
    return result


def single_mut_disulphide(dis_list):
    result = []
    not_result = []
    for dis in dis_list:
        if dis[2] == "\"CYS\"" or dis[5] == "\"CYS\"":
            result.append(dis)
        else:
            not_result.append(dis)
    return result, not_result


def propose_single_mut(single_mut_dis_list, structure_id,
start_number):
    mut = []
    for dis in single_mut_dis_list:
        if dis[2] != dis[5]:
            if dis[2] != "\"CYS\"":
                new = True
                pair_chain = dis[3]
                pair_no = dis[4]
                for diS in single_mut_dis_list:
                    if diS[0] == pair_chain and diS[1] == pair_no and
diS[2] == diS[5]:
                        new = False
                    if diS[3] == pair_chain and diS[4] == pair_no and
diS[2] == diS[5]:
                        new = False
                if new:
                    new_mut = PointMutation(three_to_one[dis[2]], "C",
dis[1], dis[0], structure_id, start_number)
                    new_mut.disulphide = ["single", dis[6]]
                    mut.append(new_mut)
            else:
                new = True
                pair_chain = dis[0]
                pair_no = dis[1]
                for diS in single_mut_dis_list:
                    if diS[0] == pair_chain and diS[1] == pair_no and
diS[2] == diS[5]:
                        new = False
                    if diS[3] == pair_chain and diS[4] == pair_no and
```

```
diS[2] == diS[5]:
                            new = False
                if new:
                    new_mut = PointMutation(three_to_one[dis[5]], "C",
dis[4], dis[3], structure_id, start_number)
                    new_mut.disulphide = dis[6]
                    mut.append(new_mut)
    return mut


def evaluate_mut_dis(single_mut_dis_list, mut):
    value = 0
    if mut.original != "C" and mut.mutated == "C":
        for dis in single_mut_dis_list:
            if (int(mut.position) == int(dis[1]) and dis[0] ==
"\"%s\"" % mut.chain) or\
                    (int(mut.position) == int(dis[4]) and dis[3] ==
"\"%s\"" % mut.chain):
                if float(dis[6]) > value:
                    value = float(dis[6])
    if mut.original == "C" and mut.mutated != "C":
        for dis in single_mut_dis_list:
            if (int(mut.position) == int(dis[1]) and dis[0] ==
"\"%s\"" % mut.chain and dis[5] == "\"CYS\"") or \
                    (int(mut.position) == int(dis[4]) and dis[3] ==
"\"%s\"" % mut.chain and dis[2] == "\"CYS\""):
                if float(dis[6]) > value and dis[0] == dis[3]:
                    value = float(dis[6])
        value = -value
    return value


def propose_double_mut(double_mut_dis_list, structure_id,
start_number):
    mut = []
    for dis in double_mut_dis_list:
        if three_to_one[dis[2]] in "ASTDENQM" and three_to_one[dis[5]]
in "ASTDENQM":
            mutation_1 = PointMutation(three_to_one[dis[2]], "C",
dis[1], dis[0], structure_id, start_number)
            mutation_1.disulphide = ["double", dis[6]]
            mutation_2 = PointMutation(three_to_one[dis[5]], "C",
dis[4], dis[3], structure_id, start_number)
            mutation_2.disulphide = ["double", dis[6]]
            new_mut = [mutation_1, mutation_2]
            mut.append(new_mut)
    return mut


def mut_dis_list(structure_id):
    result = read_dbd_file(structure_id, "txt")
    result, not_res = single_mut_disulphide(result)
    return result


def main_dis_mutation(structure_id, start_number):
    single_dis = read_dbd_file(structure_id, "txt")
    single_dis, double_dis = single_mut_disulphide(single_dis)
    single mut = propose single mut(single dis, structure id,
start_number)
    double_mut = propose_double_mut(double_dis, structure_id,
```

```
start_number)
    return single_dis, single_mut, double_mut
```

Exposure (clean_exposure.py)

```
from clean_alpha_helix import get_relative_exposure, exposure_dict
from mutation import PointMutation


def get_relative_exposure_clean(structure_id):
    full = get_relative_exposure(structure_id)
    new = []
    previous = ["chain", "aa", 0, "exp"]
    ok = False
    for item in full:
        if item[2] - previous[2] == 1:
            if ok:
                new.append(previous)
            ok = True
        else:
            ok = False
        previous = item
    return new


def propose_exp_mutations(rel_exp_list, structure_id, start_number):
    mut_list = []
    for item in rel_exp_list:
        if item[1] in "QN" and item[3] <= 0.15:
            mutation = PointMutation(item[1], "L", item[2], item[0],
structure_id, start_number)
            value = (1 - item[3]) * (0.0276 *
(exposure_dict[mutation.mutated] - exposure_dict["A"]) +
                                     0.0072 * (exposure_dict["A"] -
exposure_dict[mutation.original]))
            mutation.exposure = value
            mut_list.append(mutation)
        elif item[1] in "VLIFM" and item[3] > 1:
            if item[1] in "LIM":
                mutated = "Q"
            elif item[1] == "F":
                mutated = "Y"
            elif item[1] == "V":
                mutated = "N"
            mutation = PointMutation(item[1], mutated, item[2],
item[0], structure_id, start_number)
            value = (1 - item[3]) * (0.0276 * (exposure_dict["A"] -
exposure_dict[mutation.original]) +
                                     0.0072 *
(exposure_dict[mutation.mutated] - exposure_dict["A"]))
            mutation.exposure = value
            mut_list.append(mutation)
    return mut_list


def evaluate_exp_mutations(rel_exp_list, mutation):
    value = 0
    rel_exp = 0.75
    apolar = "VLIFM"
    polar = "STYNQ"
    for item in rel_exp_list:
```

```
        if mutation.position == item[2] and mutation.chain == item[0]:
            rel_exp = item[3]
            break
    if rel_exp > 1 or rel_exp <= 0.25:
        if mutation.mutated in polar and mutation.original in apolar:
            value = (1 - rel_exp) * (0.0276 * (exposure_dict["A"] -
exposure_dict[mutation.original]) +
                                     0.0072 *
(exposure_dict[mutation.mutated] - exposure_dict["A"]))
        if mutation.original in polar and mutation.mutated in apolar:
            value = (1 - rel_exp) * (0.0276 *
(exposure_dict[mutation.mutated] - exposure_dict["A"]) +
                                     0.0072 * (exposure_dict["A"] -
exposure_dict[mutation.original]))
    return value


def main_exp_mutation(structure_id, start_number):
    exposure = get_relative_exposure_clean(structure_id)
    exp_mut = propose_exp_mutations(exposure, structure_id,
start_number)
    return exposure, exp_mut
```

Acidic hydrogen-bonded residues (clean_acid_hbonds.py)

```
from clean_alpha_helix import relevant_acid_hbonds
from clean_exposure import get_relative_exposure_clean
from mutation import PointMutation


def propose_acid_hbond_mutation(structure_id, start_number):
    residues = get_relative_exposure_clean(structure_id)
    mut_list = []
    bonds = relevant_acid_hbonds(structure_id)
    for item in bonds:
        if item[2] in "DE":
            exposure = 0
            for residue in residues:
                if residue[0] == item[0] and residue[2] ==
int(item[1]):
                    exposure = residue[3]
                    break
            if exposure >= 0.85:
                if item[2] == "D":
                    original = "D"
                    mutated = "N"
                    value = 1
                if item[2] == "E":
                    original = "E"
                    mutated = "Q"
                    value = 1
                mutation = PointMutation(original, mutated,
int(item[1]), item[0], structure_id, start_number)
                mutation.acid_bonds = value
                mut_list.append(mutation)
    return bonds, mut_list


def evaluate_acid_hbond_mutation(bonds, residues, mutation):
    value = 0
    code = [mutation.chain, str(mutation.position), mutation.original]
```

```
    if code in bonds:
        exposure = 0
        for residue in residues:
            if residue[0] == mutation.chain and residue[2] ==
mutation.position:
                exposure = residue[3]
                break
        if exposure >= 0.85:
            if mutation.original in "DE" and mutation.mutated in "QN":
                value = 1
            if mutation.original in "QN" and mutation.mutated in "DE":
                value = -1
    return value
```

Cavities and steric clashes (clean_cavity.py)

```
import subprocess
from clean_consensus import read_sequence
from mutation import PointMutation


def run_betavoid(structure_id):
    input_dir = "running_projects/%s/pdb%s.ent" % (structure_id,
structure_id)
    output = subprocess.check_output(["/usr/bin/BetaVoid/BetaVoid64",
"-t", "lr", "-r", "1.4", "-b", "false", input_dir])
    output = str(output)
    output = output.split("\\n")
    return output


def analysis_cavities(structure_id):
    output = run_betavoid(structure_id)
    for line in output:
        if "TOTAL-VOID-VOLUME" in line:
            result = line.split()[1]
            break
    return float(result)


def scwrl_wt(structure_id, chain_pdb):
    file = open("running_projects/%s/pdb%s.txt" % (structure_id,
structure_id), "w")
    for chain in chain_pdb:
        for item in chain_pdb[chain]:
            seq = str(chain).lower()
            file.write(seq + "\n")
    file.close()
    if structure_id == "1bni":
        directory = "running_projects/1a2p/pdb1a2p.ent"
    else:
        directory = "running_projects/%s/pdb%s.ent" % (structure_id,
structure_id)
    seq_dir = "running_projects/%s/pdb%s.txt" % (structure_id,
structure_id)
    out_dir = "running_projects/%s/pdbSCWRL.ent" % structure_id
    output = subprocess.check_output(["/usr/bin/scwrl4/Scwrl4", "-i",
directory, "-o", out_dir, "-s", seq_dir, "-h"])
    output = str(output)
    output = output.split("\\n")
    energy = 0
```

```
    for line in output:
        if "Total minimal energy of the graph" in line:
            line = line.split()
            energy = line[len(line)-1]
    return float(energy)


def scwrl_mut(structure_id, mutation, wt_energy, chain_pdb):
    file = open("running_projects/%s/pdb%sSCWRL%s%i%s.txt" %
(structure_id, structure_id, mutation.original,
mutation.position, mutation.mutated), "w")
    for chain in chain_pdb:
        for item in chain_pdb[chain]:
            seq = str(chain).lower()
            if item == mutation.chain:
                index = mutation.seq_pos - 1
                seq = seq[0:index] + mutation.mutated +
seq[index+1:len(seq)]
            file.write(seq + "\n")
    file.close()
    if structure_id == "1bni":
        directory = "running_projects/1a2p/pdb1a2p.ent"
    else:
        directory = "running_projects/%s/pdb%s.ent" % (structure_id,
structure_id)
    seq_dir = "running_projects/%s/pdb%sSCWRL%s%i%s.txt" %
(structure_id, structure_id, mutation.original,
mutation.position, mutation.mutated)
    out_dir = "running_projects/%s/pdbSCWRL%s%i%s.ent" %
(structure_id, mutation.original,
mutation.position, mutation.mutated)
    output = subprocess.check_output(["/usr/bin/scwrl4/Scwrl4", "-i",
directory, "-o", out_dir, "-s", seq_dir, "-h"])
    output = str(output)
    output = output.split("\\n")
    energy = 0
    for line in output:
        if "Total minimal energy of the graph" in line:
            line = line.split()
            energy = line[len(line) - 1]
    energy = float(wt_energy) - float(energy)
    return float(energy)


def run_betavoid_scwrl(structure_id):
    input_dir = "running_projects/%s/pdbSCWRL.ent" % structure_id
    output = subprocess.check_output(["/usr/bin/BetaVoid/BetaVoid64",
"-t", "lr", "-r", "1.4", "-b", "false", input_dir])
    output = str(output)
    output = output.split("\\n")
    return output


def analysis_cavities_scwrl(structure_id):
    output = run_betavoid_scwrl(structure_id)
    for line in output:
        if "TOTAL-VOID-VOLUME" in line:
            result = line.split()[1]
            break
    return float(result)
```

```
def run_betavoid_mutation(structure_id, mutation):
    input_dir = "running_projects/%s/pdbSCWRL%s%i%s.ent" %
(structure id, mutation.original,
mutation.position, mutation.mutated)
    output = subprocess.check_output(["/usr/bin/BetaVoid/BetaVoid64",
"-t", "lr", "-r", "1.4", "-b", "false", input_dir])
    output = str(output)
    output = output.split("\\n")
    return output


def analysis_cavities_mutation(structure_id, mutation,
wt_cavity_scwrl):
    output = run_betavoid_mutation(structure_id, mutation)
    for line in output:
        if "TOTAL-VOID-VOLUME" in line:
            result = float(line.split()[1])
            break
    result -= float(wt_cavity_scwrl)
    return float(result)


def get_wt_data(structure_id, chain_pdb):
    energy = scwrl_wt(structure_id, chain_pdb)
    cavity = analysis_cavities_scwrl(structure_id)
    return energy, cavity


def evaluate_cavity_mutations(structure_id, mut, wt_energy, wt_cavity,
chain_pdb):
    energy = scwrl_mut(structure_id, mut, wt_energy, chain_pdb)
    cavity = analysis_cavities_mutation(structure_id, mut, wt_cavity)
    return energy, cavity
```

Scoring model (clean_scoring_model.py)

```
from mutation import PointMutation
from sklearn.linear_model import LogisticRegression



def get_data(name):
    file = open("first_test/%s_results_evaluation.txt" % name)
    file.readline()
    train_file = open("first_test/%s_train.txt" % name, "w")
    test_file = open("first_test/%s_test.txt" % name, "w")
    Xtrain = []
    Xtest = []
    ytrain = []
    ytest = []
    for line in file.readlines():
        data = []
        line = line.split()
        if float(line[15]) > 1 and float(line[16]) < float(line[15]):
            helix = line[8]
            if helix == "X":
                helix = 0
            data = [float(line[5]), float(line[6]), float(helix),
float(line[10]), float(line[11]), float(line[12]),
                    float(line[13]), float(line[14]), 1]
```

```
        elif float(line[15]) < 1 and float(line[15]) + float(line[16])
< 1:
            data = [float(line[5]), float(line[6]), float(helix),
float(line[10]), float(line[11]), float(line[12]),
                    float(line[13]), float(line[14]), 0]
        if data != []:
            if line[17] == "train":
                Xtrain.append(data[:8])
                ytrain.append(data[8])
                str_data = []
                for item in data:
                    str_data.append(str(item))
                train_file.write(line[0] + " " + " ".join(str_data) +
" " + line[15] + "\n")
            elif line[17] == "test":
                Xtest.append(data[:8])
                ytest.append(data[8])
                str_data = []
                for item in data:
                    str_data.append(str(item))
                test_file.write(line[0] + " " + " ".join(str_data) + "
" + line[15] + "\n")
    return Xtrain, Xtest, ytrain, ytest


def train_model():
    Xtrain, Xtest, ytrain, ytest = get_data("full")
    clf = LogisticRegression()
    clf.fit(Xtest, ytest)
    return clf


def evaluate_mutation(mutation):
    Xtrain, Xtest, ytrain, ytest = get_data("full")
    clf = LogisticRegression()
    clf.fit(Xtest, ytest)
    anc = mutation.ancestral
    cons = mutation.consensus
    hel = mutation.helix[1]
    if hel == "X":
        hel = 0
    dis = mutation.disulphide
    hbo = mutation.acid_bonds
    exp = mutation.exposure
    ene = mutation.energy
    cav = mutation.cavities
    data = [[anc, cons, hel, dis, hbo, exp, ene, cav]]
    score = clf.predict_proba(data)
    score = score[0][1]
    return score
```

Main program (prototype.py)

```
from clean_consensus import evaluate_consensus_mutation,
evaluate_ancestral_mutation, ancestral_and_consensus
from clean_alpha_helix import evaluate_helix_mutation,
main_helix_mutation
from clean_exposure import main_exp_mutation, evaluate_exp_mutations
from clean_acid_hbonds import evaluate_acid_hbond_mutation,
propose_acid_hbond_mutation
from clean_disulphide import evaluate_mut_dis, main_dis_mutation
from clean_cavity import get_wt_data, evaluate_cavity_mutations
from clean_scoring_model import evaluate_mutation
from clean_retrieval import retrieve_PDB_file
from shutil import rmtree
import subprocess
import HTML
color_dict = {"+": "green", "0": "yellow", "-": "red", "*": "green"}
start_dict = {"1a43": 145, "1ag2": 124, "1amq": 5, "1aon": 2, "1av1":
43, "1axb": 26, "1b26": 9, "1b5m": 3,
              "1cah": 2, "1cey": 2, "1dil": 2, "1fc1": 223, "1fnf":
1142, "1ftg": 2, "1g6n": 0, "1h7m": -2,
              "1ir3": 978, "1kdx": 586, "1ls4": -15, "1mbg": 90,
"1mjc": 2, "1msi": 0, "1qqv": 10, "1sak": 319,
              "1thq": 0, "1tit": -8, "1ycc": -5, "1yea": -9, "2hpr":
2, "2q98": 9, "2zta": 0, "3bls": 4,
              "3pgk": 0, "4blm": 26}


def propose_mutations(structure_id):
    start_number = 1
    if structure_id in start_dict:
        start_number = start_dict[structure_id]
    chain_pdb, ancestral, anc_mut, alignment, cons_mut =
ancestral_and_consensus(structure_id, start_number)
    helix, hel_mut = main_helix_mutation(structure_id, start_number)
    exposure, exp_mut = main_exp_mutation(structure_id, start_number)
    acid_hbonds, acid_mut = propose_acid_hbond_mutation(structure_id,
start_number)
    dis, single_dis_mut, double_dis_mut =
main_dis_mutation(structure_id, start_number)
    wt_energy, wt_cavity = get_wt_data(structure_id, chain_pdb)
    data = [alignment, ancestral, chain_pdb, helix, exposure,
acid_hbonds, dis, wt_energy, wt_cavity]
    single_mut = [anc_mut, cons_mut, hel_mut, exp_mut, acid_mut,
single_dis_mut]
    return data, single_mut, double_dis_mut


def merge_single_mut(single_mut):
    anc = single_mut[0]
    cons = single_mut[1]
    hel = single_mut[2]
    dis = single_mut[3]
    hbo = single_mut[4]
    exp = single_mut[5]
    ac = []
    hd = []
    he = []
    achd = []
    total = []
    if anc == []:
        ac = cons
```

```
    elif cons == []:
        ac = anc
    else:
        for item in cons:
            if item.chain == "A":
                mut = item
                for i in range(len(anc)):
                    mut2 = anc[i]
                    if mut2 != 0:
                        if mut.position == mut2.position and
mut.original == mut2.original and mut.mutated == mut2.mutated and
mut.chain == mut2.chain:
                            mut.ancestral = mut2.ancestral
                            anc[i] = 0
                            break
                ac.append(mut)
        for item in anc:
            if item != 0 and item.chain == "A":
                ac.append(item)
    if hel == []:
        hd = dis
    elif dis == []:
        hd = hel
    else:
        for item in dis:
            if item.chain == "A":
                mut = item
                for i in range(len(hel)):
                    mut2 = hel[i]
                    if mut2 != 0:
                        if mut.position == mut2.position and
mut.original == mut2.original and mut.mutated == mut2.mutated and
mut.chain == mut2.chain:
                            mut.helix = mut2.helix
                            hel[i] = 0
                            break
                hd.append(mut)
        for item in hel:
            if item != 0 and item.chain == "A":
                hd.append(item)
    if hbo == []:
        he = exp
    elif exp == []:
        he = hbo
    else:
        for item in exp:
            if item.chain == "A":
                mut = item
                for i in range(len(hbo)):
                    mut2 = hbo[i]
                    if mut2 != 0:
                        if mut.position == mut2.position and
mut.original == mut2.original and mut.mutated == mut2.mutated and
mut.chain == mut2.chain:
                            mut.acid_bonds = mut2.acid_bonds
                            hbo[i] = 0
                            break
                he.append(mut)
        for item in hbo:
            if item != 0 and item.chain == "A":
                he.append(item)
```

```
    if hd == []:
        achd = ac
    elif ac == []:
        achd = hd
    else:
        for item in hd:
            if item.chain == "A":
                mut = item
                for i in range(len(ac)):
                    mut2 = ac[i]
                    if mut2 != 0:
                        if mut.position == mut2.position and
mut.original == mut2.original and mut.mutated == mut2.mutated and
mut.chain == mut2.chain:
                            mut.ancestral = mut2.ancestral
                            mut.consensus = mut2.consensus
                            ac[i] = 0
                            break
                achd.append(mut)
        for item in ac:
            if item != 0 and item.chain == "A":
                achd.append(item)
    if achd == []:
        total = he
    elif he == []:
        total = achd
    else:
        for item in he:
            if item.chain == "A":
                mut = item
                for i in range(len(achd)):
                    mut2 = achd[i]
                    if mut2 != 0:
                        if mut.position == mut2.position and
mut.original == mut2.original and mut.mutated == mut2.mutated and
mut.chain == mut2.chain:
                            mut.ancestral = mut2.ancestral
                            mut.consensus = mut2.consensus
                            mut.helix = mut2.helix
                            mut.disulphide = mut2.disulphide
                            achd[i] = 0
                            break
                total.append(mut)
        for item in achd:
            if item != 0 and item.chain == "A":
                total.append(item)
    return total


def get_proposed_by(mut):
    proposed = []
    if mut.ancestral != "":
        proposed.append(0)
    if mut.consensus != "":
        proposed.append(1)
    if mut.helix != []:
        proposed.append(2)
    if mut.disulphide != "":
        proposed.append(3)
    if mut.acid_bonds != "":
        proposed.append(4)
```

```
    if mut.exposure != "":
        proposed.append(5)
    mut.proposed_by = proposed
    return mut


def evaluate_single_mutation(data, mutation):
    alignment = data[0]
    ancestral = data[1]
    chain_pdb = data[2]
    helix = data[3]
    exposure = data[4]
    acid_hbonds = data[5]
    dis = data[6]
    wt_energy = data[7]
    wt_cavity = data[8]
    mutation.energy, mutation.cavities =
evaluate_cavity_mutations(mutation.structure_id, mutation, wt_energy,

wt_cavity, chain_pdb)
    if 0 not in mutation.proposed_by:
        mutation.ancestral = evaluate_ancestral_mutation(ancestral,
mutation)
    if 1 not in mutation.proposed_by:
        mutation.consensus = evaluate_consensus_mutation(alignment,
mutation)
    if 2 not in mutation.proposed_by:
        mutation.helix = evaluate_helix_mutation(helix, mutation)
    if 3 not in mutation.proposed_by:
        mutation.disulphide = evaluate_mut_dis(dis, mutation)
    if 4 not in mutation.proposed_by:
        mutation.acid_bonds =
evaluate_acid_hbond_mutation(acid_hbonds, exposure, mutation)
    if 5 not in mutation.proposed_by:
        mutation.exposure = evaluate_exp_mutations(exposure, mutation)
    return mutation


def evaluate_double_mutation(data, d_mutation):
    new = []
    alignment = data[0]
    ancestral = data[1]
    chain_pdb = data[2]
    helix = data[3]
    exposure = data[4]
    acid_hbonds = data[5]
    dis = data[6]
    wt_energy = data[7]
    wt_cavity = data[8]
    for mutation in d_mutation:
        mutation.energy, mutation.cavities =
evaluate_cavity_mutations(mutation.structure_id, mutation, wt_energy,

wt cavity, chain pdb)
        mutation.ancestral = evaluate_ancestral_mutation(ancestral,
mutation)
        mutation.consensus = evaluate_consensus_mutation(alignment,
mutation)
        mutation.helix = evaluate helix mutation(helix, mutation)
        mutation.disulphide = evaluate_mut_dis(dis, mutation)
        mutation.acid_bonds =
```

```
evaluate_acid_hbond_mutation(acid_hbonds, exposure, mutation)
        mutation.exposure = evaluate_exp_mutations(exposure, mutation)
        new.append(mutation)
    return new


def order_mutations(score, mut):
    new_score = []
    new_mut = []
    for i in range(len(score)):
        max_score = max(score)
        index = score.index(max_score)
        new_score.append(score[index])
        new_mut.append(mut[index])
        mut[index] = 0
        score[index] = 0
    return new_score, new_mut


def main_function():
    structure_id = input("PDB code:").lower()
    retrieve_PDB_file(structure_id)
    data, single_mut, double_mut = propose_mutations(structure_id)
    single_mut = merge_single_mut(single_mut)
    full_single = []
    for mut in single_mut:
        if mut.chain == "A":
            mut = get_proposed_by(mut)
            full_single.append(evaluate_single_mutation(data, mut))
    full_double = []
    for mut in double_mut:
        if mut[0].chain == "A" and mut[1].chain == "A":
            full_double.append(evaluate_double_mutation(data, mut))
    single_mut = []
    single_score = []
    for mut in full_single:
        score = evaluate_mutation(mut)
        single_score.append(score)
        mut.score = score
        single_mut.append(mut)
    double_mut = []
    double_score = []
    for mut in full_double:
        mut1 = mut[0]
        mut2 = mut[1]
        score1 = evaluate_mutation(mut1)
        mut1.score = score1
        score2 = evaluate_mutation(mut2)
        mut2.score = score2
        score = (score1+score2)/2.0
        double_mut.append([mut1, mut2])
        double_score.append(score)
    double_score, double_mut = order_mutations(double_score,
double mut)
    single_score, single_mut = order_mutations(single_score,
single_mut)
    file = open("prototype_results/%s.txt" % structure_id, "w")
    file.write(" ".join(["Mutation", "Probability", "Proposed_by"
"Anc", "Cons", "Hel", "DiS", "AcB", "Exp", "Ene", "Cav"])+"\n")
    table = HTML.Table(header_row=["Mutation", "Probability", "Anc",
"Cons", "Hel", "DiS", "AcB", "Exp", "Ene", "Cav"],
```

```
                        col_align=["center", "center", "center",
"center", "center", "center", "center", "center", "center",
                                    "center", "center"])
    for i in range(len(single_score)):
        item = single_mut[i]
        ancestral = item.ancestral
        if ancestral > 0:
            ancestral = "+"
        elif ancestral == 0:
            ancestral = "0"
        consensus = item.consensus
        if consensus < 0:
            consensus = "-"
        elif consensus > 0:
            consensus = "+"
        elif consensus == 0:
            consensus = "0"
        helix = item.helix[1]
        if helix == "X":
            thelix = 0
            helix = "0"
        else:
            helix = float(helix)
            if helix == 0:
                thelix = 0
                helix = "0"
            elif helix < 0:
                thelix = helix
                helix = "-"
            elif helix > 0:
                thelix = helix
                helix = "+"
        disulphide = item.disulphide
        if disulphide < 0:
            disulphide = "-"
        elif disulphide > 0:
            disulphide = "+"
        elif disulphide == 0:
            disulphide = "0"
        acid = item.acid_bonds
        if acid < 0:
            acid = "-"
        elif acid > 0:
            acid = "+"
        elif acid == 0:
            acid = "0"
        exp = item.exposure
        if exp < 0:
            exp = "-"
        elif exp > 0:
            exp = "+"
        elif exp == 0:
            exp = "0"
        energy = item.energy
        if energy < 0:
            energy = "-"
        elif energy > 0:
            energy = "+"
        elif energy == 0:
            energy = "0"
        cavities = item.cavities
```

```
        if cavities < 0:
            cavities = "-"
        elif cavities > 0:
            cavities = "+"
        elif cavities == 0:
            cavities = "0"
        proposed = item.proposed_by
        if 0 in proposed:
            ancestral = "*"
        if 1 in proposed:
            consensus = "*"
        if 2 in proposed:
            helix = "*"
        if 3 in proposed:
            disulphide = "*"
        if 4 in proposed:
            acid = "*"
        if 5 in proposed:
            exp = "*"
        mut = HTML.TableCell(str(item))
        prob = HTML.TableCell(str(round(100*single_score[i], 1))+"%")
        ancestral = HTML.TableCell(ancestral,
bgcolor=color_dict[ancestral])
        consensus = HTML.TableCell(consensus,
bgcolor=color_dict[consensus])
        helix = HTML.TableCell(helix, bgcolor=color_dict[helix])
        disulphide = HTML.TableCell(disulphide,
bgcolor=color_dict[disulphide])
        acid = HTML.TableCell(acid, bgcolor=color_dict[acid])
        exp = HTML.TableCell(exp, bgcolor=color_dict[exp])
        energy = HTML.TableCell(energy, bgcolor=color_dict[energy])
        cavities = HTML.TableCell(cavities,
bgcolor=color_dict[cavities])
        table.rows.append([mut, prob, ancestral, consensus, helix,
disulphide, acid, exp, energy, cavities])
        file.write(" ".join(["_".join(str(item).split()),
str(single_score[i]), "".join(str(item.proposed_by).split()),
                            str(item.ancestral), str(item.consensus),
str(thelix), str(item.disulphide),
                            str(item.acid_bonds), str(item.exposure),
str(item.energy), str(item.cavities)])+"\n")
    html_file = open("prototype_results/%s.html" % structure_id, "w")
    html_file.write("<!DOCTYPE HTML>\n<html>\n<head>\n\n<title>Results
for structure %s</title>\n</head>\n\n"
                    "<body>\n<h1>Single mutation results</h1>\n" %
structure_id)
    html_file.write(str(table))

file.write("\n___________DOUBLE_MUTATIONS_____________________________\n"
)
    file.write(" ".join(["Mutation1", "Mutation2", "Probability",
"Proposed_by", "Anc1", "Anc2", "Cons1", "Cons2",
                         "Hel1", "Hel2", "DiS1", "DiS2", "AcB1",
"AcB2", "Exp1", "Exp2", "Ene1", "Ene2",
                         "Cav1", "Cav2"])+"\n")
    table = HTML.Table(header_row=["Mutation1", "Mutation2",
"Probability", "Anc1", "Anc2", "Cons1", "Cons2",
                                   "Hel1", "Hel2", "DiS1", "DiS2",
"AcB1", "AcB2", "Exp1", "Exp2", "Ene1", "Ene2",
                                   "Cav1", "Cav2"],
                       col_align=["center", "center", "center",
```

```
"center", "center", "center", "center", "center",
                                    "center", "center", "center",
"center", "center", "center", "center", "center",
                                    "center", "center", "center"])
    for i in range(len(double_score)):
        item1 = double_mut[i][0]
        item2 = double_mut[i][1]
        ancestral1 = item1.ancestral
        if ancestral1 > 0:
            ancestral1 = "+"
        elif ancestral1 == 0:
            ancestral1 = "0"
        consensus1 = item1.consensus
        if consensus1 < 0:
            consensus1 = "-"
        elif consensus1 > 0:
            consensus1 = "+"
        elif consensus1 == 0:
            consensus1 = "0"
        helix1 = item1.helix[1]
        if helix1 == "X":
            thelix1 = 0
            helix1 = "0"
        else:
            helix1 = float(helix1)
            if helix1 == 0:
                thelix1 = 0
                helix1 = "0"
            elif helix1 < 0:
                thelix1 = helix1
                helix1 = "-"
            elif helix1 > 0:
                thelix1 = helix1
                helix1 = "+"
        disulphide1 = "*"
        acid1 = item1.acid_bonds
        if acid1 < 0:
            acid1 = "-"
        elif acid1 > 0:
            acid1 = "+"
        elif acid1 == 0:
            acid1 = "0"
        exp1 = item1.exposure
        if exp1 < 0:
            exp1 = "-"
        elif exp1 > 0:
            exp1 = "+"
        elif exp1 == 0:
            exp1 = "0"
        energy1 = item1.energy
        if energy1 < 0:
            energy1 = "-"
        elif energy1 > 0:
            energy1 = "+"
        elif energy1 == 0:
            energy1 = "0"
        cavities1 = item1.cavities
        if cavities1 < 0:
            cavities1 = "-"
        elif cavities1 > 0:
            cavities1 = "+"
```

```
elif cavities1 == 0:
    cavities1 = "0"
ancestral2 = item2.ancestral
if ancestral2 > 0:
    ancestral2 = "+"
elif ancestral2 == 0:
    ancestral2 = "0"
consensus2 = item2.consensus
if consensus2 < 0:
    consensus2 = "-"
elif consensus2 > 0:
    consensus2 = "+"
elif consensus2 == 0:
    consensus2 = "0"
helix2 = item2.helix[1]
if helix2 == "X":
    thelix2 = 0
    helix2 = "0"
else:
    helix2 = float(helix2)
    if helix2 == 0:
        thelix2 = 0
        helix2 = "0"
    elif helix2 < 0:
        thelix2 = helix2
        helix2 = "-"
    elif helix2 > 0:
        thelix2 = helix2
        helix2 = "+"
disulphide2 = "*"
acid2 = item2.acid_bonds
if acid2 < 0:
    acid2 = "-"
elif acid2 > 0:
    acid2 = "+"
elif acid2 == 0:
    acid2 = "0"
exp2 = item2.exposure
if exp2 < 0:
    exp2 = "-"
elif exp2 > 0:
    exp2 = "+"
elif exp2 == 0:
    exp2 = "0"
energy2 = item2.energy
if energy2 < 0:
    energy2 = "-"
elif energy2 > 0:
    energy2 = "+"
elif energy2 == 0:
    energy2 = "0"
cavities2 = item2.cavities
if cavities2 < 0:
    cavities2 = "-"
elif cavities2 > 0:
    cavities2 = "+"
elif cavities2 == 0:
    cavities2 = "0"
mut1 = HTML.TableCell(str(item1))
mut2 = HTML.TableCell(str(item2))
prob = HTML.TableCell(str(round(100 * double_score[i],
```

```
1))+"%")
        ancestral1 = HTML.TableCell(ancestral1,
bgcolor=color_dict[ancestral1])
        consensus1 = HTML.TableCell(consensus1,
bgcolor=color_dict[consensus1])
        helix1 = HTML.TableCell(helix1, bgcolor=color_dict[helix1])
        disulphide1 = HTML.TableCell(disulphide1,
bgcolor=color_dict[disulphide1])
        acid1 = HTML.TableCell(acid1, bgcolor=color_dict[acid1])
        exp1 = HTML.TableCell(exp1, bgcolor=color_dict[exp1])
        energy1 = HTML.TableCell(energy1, bgcolor=color_dict[energy1])
        cavities1 = HTML.TableCell(cavities1,
bgcolor=color_dict[cavities1])
        ancestral2 = HTML.TableCell(ancestral2,
bgcolor=color_dict[ancestral2])
        consensus2 = HTML.TableCell(consensus2,
bgcolor=color_dict[consensus2])
        helix2 = HTML.TableCell(helix2, bgcolor=color_dict[helix2])
        disulphide2 = HTML.TableCell(disulphide2,
bgcolor=color_dict[disulphide2])
        acid2 = HTML.TableCell(acid2, bgcolor=color_dict[acid2])
        exp2 = HTML.TableCell(exp2, bgcolor=color_dict[exp2])
        energy2 = HTML.TableCell(energy2, bgcolor=color_dict[energy2])
        cavities2 = HTML.TableCell(cavities2,
bgcolor=color_dict[cavities2])
        table.rows.append([mut1, mut2, prob, ancestral1, ancestral2,
consensus1, consensus2, helix1, helix2, disulphide1,
                           disulphide2, acid1, acid2, exp1, exp2,
energy1, energy2, cavities1, cavities2])
        file.write(" ".join(["_".join(str(item1).split()),
"_".join(str(item2).split()), str(double_score[i]),
                             "3", str(item1.ancestral),
str(item2.ancestral), str(item1.consensus), str(item2.consensus),
                             str(thelix1), str(thelix2),
str(item1.disulphide), str(item2.disulphide),
                             str(item1.acid_bonds),
str(item2.acid_bonds), str(item1.exposure), str(item2.exposure),
                             str(item1.energy), str(item2.energy),
str(item1.cavities), str(item2.cavities)]) + "\n")
    html_file.write("\n<h1>Double mutation results</h1>\n")
    html_file.write(str(table))
    html_file.write("\n<h2>Color code legend</h2>\n")
    table = HTML.Table(header_row=["Code", "Meaning"],
col_align=["center", "center"])
    color_mean = {"*": "Proposed by this module", "+": "Positive score
for this module",
                  "0": "Neutral score for this module", "-": "Negative
score for this module"}
    for item in color_dict:
        meaning = HTML.TableCell(color_mean[item])
        item = HTML.TableCell(item, bgcolor=color_dict[item])
        table.rows.append([item, meaning])
    html_file.write(str(table))
    table = HTML.Table(header_row=["Module", "Full name"],
col_align=["center", "center"])
    module_dict = {"Anc": "Reconstruction of the ancestral sequence",
"Cons": "Prediction of the consensus sequence",
                   "Hel": "Study of alpha-helix stability", "DiS":
"Study of disulphide bonds",
                   "AcB": "Study of exposed acidic hydrogen-bonded
residues",
```

```
                    "Exp": "Study of the exposure and polarity of the
residues",
                    "Ene": "Study of the minimum energy of the protein
structure",
                    "Cav": "Study of the volume of internal cavities in
the protein"}
    for item in module_dict:
        table.rows.append([item, module_dict[item]])
    html_file.write("\n<h2>Modules legend</h2>\n")
    html_file.write(str(table))
    html_file.write("\n</body>\n</html>")
    html_file.close()
    subprocess.call(["firefox",
"/home/elhectro2/Documentos/TFM/Programming_TFM/prototype_results/%s.h
tml" %
                     structure_id])
    rmtree("running_projects/%s" % structure_id)

main_function()
```

Appendix B. Dataset for alpha-helix mutation evaluation

Internal residues

Residue	kcal/mol	Residue	kcal/mol
A	0	**L**	0.19
R	0.09	**K**	0.2
N	0.44	**M**	0.31
D	0.4	**F**	0.51
C	0.67	**P**	3.47
E	0.22	**S**	0.37
Q	0.21	**T**	0.47
G	0.82	**W**	0.53
H	0.54	**Y**	0.53
I	0.42	**V**	0.6

N-cap

Residue	kcal/mol	Residue	kcal/mol
A	1.1	**L**	1.18
R	1.1	**K**	1.1
N	0	**M**	1.18
D	0	**F**	1.18
C	1.18	**P**	1.54
E	0.8	**S**	0.25
Q	1.25	**T**	0.48
G	0.35	**W**	1.18
H	1.18	**Y**	1.18
I	1.18	**V**	1.18

C-cap

Residue	kcal/mol	Residue	kcal/mol
A	0.4	**L**	0.48
R	0.15	**K**	0.3
N	0	**M**	0.48
D	0.76	**F**	0.48
C	0.48	**P**	0.48
E	0.32	**S**	0.32
Q	0.32	**T**	0.32
G	0	**W**	0.48
H	0	**Y**	0.48
I	0.48	**V**	0.48

Average exposure for each residue

Residue	Å^2	Residue	Å^2
A	76.86	**L**	138.495
R	187.845	**K**	157.395
N	114.66	**M**	141.015
D	107.31	**F**	153.405
C	93.135	**P**	105.315
E	132.3	**S**	79.8
Q	132.195	**T**	97.965
G	57.015	**W**	181.86
H	135.975	**Y**	164.745
I	128.625	**V**	107.31

Appendix C. Parameter file for PAML

This is an example of a parameter file for PAML. Some comments for a better understanding of the file are introduced after the # symbol.

```
seqfile = ../../A_alignment_OK.phy
outfile = ../../PAMLA.out
treefile = ../../A_alignment_OK1.phy_phyml_tree.txt
noisy = 1  # How much output the program returns while running
verbose = 0  # Gives output while running or not
runmode = 0  # Creates a tree or starts from an existing one.
seqtype = 2  # DNA, RNA or protein
CodonFreq = 2  # Related to substitution model.
ndata = 1
clock = 1  # Molecular clock hypothesis
aaDist = 1  # Use of a substitution model or not.
aaRatefile = /bin/paml4.9e/dat/jones.dat # Location of the model
model = 2  # Type of model
NSsites = 0
icode = 0
Mgene = 0
fix_kappa = 0
kappa = 2
fix_omega = 0
omega = 0.4
fix_alpha = 1
alpha = 0
Malpha = 0
getSE = 0
RateAncestor = 1 #Giving the probabilities of each ancestor and its
sequence or not.
Small_Diff = 5e-07
cleandata = 0
fix_blength = 1
method = 0
rho = 1
fix_rho = 0
```