In [36]:
```
!pip install fair-esm torch -qq
```

In [37]:
```
!pip install Levenshtein -qq
```

In [38]:
```
!pip install transformers -q
```

In [39]:
```
!pip install gensim -q
```

In [40]:
```
import torch
from transformers import AutoTokenizer, AutoModel
from sklearn.metrics.pairwise import cosine_similarity
from Levenshtein import distance
import numpy as np

import warnings
import os

warnings.filterwarnings("ignore")

import logging
logging.getLogger("transformers").setLevel(logging.ERROR)
logging.getLogger("huggingface_hub").setLevel(logging.ERROR)
```

In [41]:
```
# 1. Load a pre-trained Protein LLM (ESM-2 is industry standard)
model_name = "facebook/esm2_t6_8M_UR50D"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

def get_embedding(text):
    inputs = tokenizer(text, return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)
    # Mean pooling of hidden states
    return outputs.last_hidden_state.mean(dim=1).numpy()

def find_mutated_motifs(sequence, target_motif, threshold=0.85):
    target_emb = get_embedding(target_motif)
    window_size = len(target_motif)
    matches = []

    # Sliding window search
    for i in range(len(sequence) - window_size + 1):
        window = sequence[i:i+window_size]
        window_emb = get_embedding(window)

        # Cosine Similarity
        sim = np.dot(target_emb, window_emb.T) / (np.linalg.norm(target_emb) * n

        if sim > threshold:
            matches.append((window, i, float(sim)))

    return sorted(matches, key=lambda x: x[2], reverse=True)

# Example Usage
seq = "MAPLGDTRVMRDIAP"
target = "RGD"
results = find_mutated_motifs(seq, target)
```

```
for motif, pos, score in results:
    print(f"Found: {motif} at pos {pos} (Confidence: {score:.2f})")
```

Loading weights: 100% 102/102 [00:00<00:00, 734.98it/s]

```
Found: MRD at pos 9 (Confidence: 0.98)
Found: DTR at pos 5 (Confidence: 0.98)
Found: LGD at pos 3 (Confidence: 0.96)
Found: PLG at pos 2 (Confidence: 0.96)
Found: TRV at pos 6 (Confidence: 0.96)
Found: GDT at pos 4 (Confidence: 0.96)
Found: RDI at pos 10 (Confidence: 0.95)
Found: VMR at pos 8 (Confidence: 0.95)
Found: RVM at pos 7 (Confidence: 0.94)
Found: APL at pos 1 (Confidence: 0.94)
Found: MAP at pos 0 (Confidence: 0.94)
Found: IAP at pos 12 (Confidence: 0.93)
Found: DIA at pos 11 (Confidence: 0.93)
```

In [42]:

```
import torch
from transformers import AutoTokenizer, EsmForMaskedLM

# Load ESM-2 (The 35M parameter version is fast and accurate)
model_name = "facebook/esm2_t6_8M_UR50D"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = EsmForMaskedLM.from_pretrained(model_name)

def check_functional_similarity(sequence, pos, original_aa, mutated_aa):
    """
    Checks if the LLM thinks a mutation at 'pos' is functionally 'ok'.
    Higher score = More likely to preserve function.
    """
    # 1. Mask the position of interest
    seq_list = list(sequence)
    seq_list[pos] = tokenizer.mask_token
    masked_seq = "".join(seq_list)

    inputs = tokenizer(masked_seq, return_tensors="pt")

    with torch.no_grad():
        logits = model(**inputs).logits

    # 2. Get probabilities for that specific position
    mask_token_index = torch.where(inputs["input_ids"] == tokenizer.mask_token_i
    mask_token_logits = logits[0, mask_token_index, :]
    probabilities = torch.nn.functional.softmax(mask_token_logits, dim=-1)

    # 3. Compare the 'mutated' residue probability vs others
    target_aa_id = tokenizer.convert_tokens_to_ids(mutated_aa)
    score = probabilities[0, target_aa_id].item()

    return score

# Example: Sequence has LGD, we want to know if it functions like RGD
my_seq = "MAPLGDTRV"
# Position of 'L' is 3. We want to see if 'L' is a good substitute for 'R'
score = check_functional_similarity(my_seq, 3, "R", "L")

print(f"Functional Probablity Score: {score:.4f}")
```

```
if score > 0.05: # Common threshold for biological relevance
    print("This mutation is likely to preserve function.")
```

Loading weights: 100% 107/107 [00:00<00:00, 1111.13it/s]

```
Functional Probablity Score: 0.0856
This mutation is likely to preserve function.
```

In [42]:

In [43]:
```
import torch
import spacy
from gensim.models import Word2Vec
from transformers import AutoTokenizer, AutoModel
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# 1. INITIALIZE NLP AND LLM
print("Loading Protein LLM (ESM-2)...")
MODEL_NAME = "facebook/esm2_t6_8M_UR50D"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
llm_model = AutoModel.from_pretrained(MODEL_NAME)

# Use spaCy for sequence structure (treating protein as a 'document')
nlp = spacy.blank("en")

# 2. HELPER FUNCTIONS
def get_kmers(text, k=3):
    """Tokenizes sequence into overlapping k-mers (NLP words)."""
    return [text[i:i+k] for i in range(len(text) - k + 1)]

def get_llm_embedding(text):
    """Converts motif string into a high-dimensional biological vector."""
    inputs = tokenizer(text, return_tensors="pt")
    with torch.no_grad():
        outputs = llm_model(**inputs)
    return outputs.last_hidden_state.mean(dim=1).numpy()

# 3. GENSIM WORD2VEC (CO-OCCURRENCE LOGIC)
# We feed it a small dataset so it learns that these motifs appear in similar co
corpus = [
    "MAGRGDVEEEMRELMREIMRDLMRDIWQRGD",
    "LGDVEEEMREIMRDLMRDI",
    "RGDLGDMRDLMRDI"
]
tokenized_corpus = [get_kmers(seq) for seq in corpus]

print("Training Gensim Word2Vec...")
gs_model = Word2Vec(vector_size=16, window=2, min_count=1)
gs_model.build_vocab(tokenized_corpus)
gs_model.train(tokenized_corpus, total_examples=gs_model.corpus_count, epochs=20

# 4. TARGET MOTIFS AND PRIORITY
# We include 'X' as a wildcard for RXGD or RGXD
targets = {
    "RGD": {"priority": "Highest", "weight": 1.0},
    "LGD": {"priority": "High", "weight": 0.9},
    "RAGD": {"priority": "Middle", "weight": 0.7}, # Example of RXGD
    "MREL": {"priority": "Highest", "weight": 1.0},
    "DTR": {"priority": "Lowest", "weight": 0.4}
```

```
}

# 5. SCANNING THE TARGET SEQUENCE
test_sequence = "MAPLGDVEEEMREIMRDLMRDIWQRGD"
found_tokens = get_kmers(test_sequence)

print(f"\n{'K-mer':<8} | {'Target':<8} | {'Priority':<10} | {'LLM Score'}")
print("-" * 50)

for token in found_tokens:
    # Get vector for current window
    token_vec = get_llm_embedding(token)

    for target, info in targets.items():
        # Get vector for the target we are looking for
        target_vec = get_llm_embedding(target)

        # Calculate Semantic Similarity (NLP Context)
        similarity = cosine_similarity(token_vec, target_vec)[0][0]

        # Threshold: if it looks like a mutation (e.g., LGD vs RGD)
        if similarity > 0.85:
            print(f"{token:<8} | {target:<8} | {info['priority']:<10} | {similar

# 6. GENSIM "SYNONYM" DISCOVERY
print("\n--- Gensim Motif Relationships ---")
try:
    similar_to_rgd = gs_model.wv.most_similar("RGD", topn=2)
    print(f"Gensim predicts these motifs are 'synonyms' based on context: {simil
except KeyError:
    print("Gensim needs more data to find synonyms.")
```

```
Loading Protein LLM (ESM-2)...
```

Loading weights: 100% 102/102 [00:00<00:00, 579.77it/s]

```
Training Gensim Word2Vec...

K-mer    | Target   | Priority   | LLM Score
-------------------------------------------------
MAP      | RGD      | Highest    | 0.9377
MAP      | LGD      | High       | 0.9315
MAP      | RAGD     | Middle     | 0.9420
MAP      | MREL     | Highest    | 0.9575
MAP      | DTR      | Lowest     | 0.9521
APL      | RGD      | Highest    | 0.9411
APL      | LGD      | High       | 0.9715
APL      | RAGD     | Middle     | 0.9464
APL      | MREL     | Highest    | 0.9522
APL      | DTR      | Lowest     | 0.9516
PLG      | RGD      | Highest    | 0.9627
PLG      | LGD      | High       | 0.9826
PLG      | RAGD     | Middle     | 0.9560
PLG      | MREL     | Highest    | 0.9655
PLG      | DTR      | Lowest     | 0.9608
LGD      | RGD      | Highest    | 0.9643
LGD      | LGD      | High       | 1.0000
LGD      | RAGD     | Middle     | 0.9592
LGD      | MREL     | Highest    | 0.9534
LGD      | DTR      | Lowest     | 0.9591
GDV      | RGD      | Highest    | 0.9655
GDV      | LGD      | High       | 0.9771
GDV      | RAGD     | Middle     | 0.9607
GDV      | MREL     | Highest    | 0.9423
GDV      | DTR      | Lowest     | 0.9499
DVE      | RGD      | Highest    | 0.9521
DVE      | LGD      | High       | 0.9783
DVE      | RAGD     | Middle     | 0.9471
DVE      | MREL     | Highest    | 0.9411
DVE      | DTR      | Lowest     | 0.9497
VEE      | RGD      | Highest    | 0.9578
VEE      | LGD      | High       | 0.9719
VEE      | RAGD     | Middle     | 0.9460
VEE      | MREL     | Highest    | 0.9523
VEE      | DTR      | Lowest     | 0.9542
EEE      | RGD      | Highest    | 0.9188
EEE      | LGD      | High       | 0.9168
EEE      | RAGD     | Middle     | 0.9010
EEE      | MREL     | Highest    | 0.9007
EEE      | DTR      | Lowest     | 0.8945
EEM      | RGD      | Highest    | 0.9130
EEM      | LGD      | High       | 0.9226
EEM      | RAGD     | Middle     | 0.8989
EEM      | MREL     | Highest    | 0.9331
EEM      | DTR      | Lowest     | 0.8983
EMR      | RGD      | Highest    | 0.9611
EMR      | LGD      | High       | 0.9364
EMR      | RAGD     | Middle     | 0.9476
EMR      | MREL     | Highest    | 0.9885
EMR      | DTR      | Lowest     | 0.9682
MRE      | RGD      | Highest    | 0.9697
MRE      | LGD      | High       | 0.9390
MRE      | RAGD     | Middle     | 0.9543
MRE      | MREL     | Highest    | 0.9900
MRE      | DTR      | Lowest     | 0.9730
REI      | RGD      | Highest    | 0.9562
```

REI	LGD	High	0.9685
REI	RAGD	Middle	0.9494
REI	MREL	Highest	0.9720
REI	DTR	Lowest	0.9697
EIM	RGD	Highest	0.9238
EIM	LGD	High	0.9598
EIM	RAGD	Middle	0.9124
EIM	MREL	Highest	0.9596
EIM	DTR	Lowest	0.9397
IMR	RGD	Highest	0.9316
IMR	LGD	High	0.9310
IMR	RAGD	Middle	0.9169
IMR	MREL	Highest	0.9751
IMR	DTR	Lowest	0.9613
MRD	RGD	Highest	0.9772
MRD	LGD	High	0.9490
MRD	RAGD	Middle	0.9654
MRD	MREL	Highest	0.9842
MRD	DTR	Lowest	0.9819
RDL	RGD	Highest	0.9712
RDL	LGD	High	0.9753
RDL	RAGD	Middle	0.9665
RDL	MREL	Highest	0.9739
RDL	DTR	Lowest	0.9767
DLM	RGD	Highest	0.9238
DLM	LGD	High	0.9691
DLM	RAGD	Middle	0.9164
DLM	MREL	Highest	0.9552
DLM	DTR	Lowest	0.9381
LMR	RGD	Highest	0.9360
LMR	LGD	High	0.9296
LMR	RAGD	Middle	0.9226
LMR	MREL	Highest	0.9811
LMR	DTR	Lowest	0.9571
MRD	RGD	Highest	0.9772
MRD	LGD	High	0.9490
MRD	RAGD	Middle	0.9654
MRD	MREL	Highest	0.9842
MRD	DTR	Lowest	0.9819
RDI	RGD	Highest	0.9529
RDI	LGD	High	0.9640
RDI	RAGD	Middle	0.9498
RDI	MREL	Highest	0.9607
RDI	DTR	Lowest	0.9718
DIW	RGD	Highest	0.9385
DIW	LGD	High	0.9633
DIW	RAGD	Middle	0.9265
DIW	MREL	Highest	0.9573
DIW	DTR	Lowest	0.9601
IWQ	RGD	Highest	0.9199
IWQ	LGD	High	0.9258
IWQ	RAGD	Middle	0.9032
IWQ	MREL	Highest	0.9517
IWQ	DTR	Lowest	0.9439
WQR	RGD	Highest	0.9424
WQR	LGD	High	0.9044
WQR	RAGD	Middle	0.9244
WQR	MREL	Highest	0.9572
WQR	DTR	Lowest	0.9499
QRG	RGD	Highest	0.9760

```
QRG      | LGD      | High       | 0.9296
QRG      | RAGD     | Middle     | 0.9651
QRG      | MREL     | Highest    | 0.9626
QRG      | DTR      | Lowest     | 0.9628
RGD      | RGD      | Highest    | 1.0000
RGD      | LGD      | High       | 0.9643
RGD      | RAGD     | Middle     | 0.9902
RGD      | MREL     | Highest    | 0.9659
RGD      | DTR      | Lowest     | 0.9761

--- Gensim Motif Relationships ---
Gensim predicts these motifs are 'synonyms' based on context: [('RDL', 0.53231936
69319153), ('MAG', 0.3427450358867645)]
```

In [43]:

In [44]:
```python
# ==============================
# NLP-Based Motif Mutation Detection
# ==============================

import numpy as np
from gensim.models import FastText
from sklearn.metrics.pairwise import cosine_similarity
from Levenshtein import distance

# ------------------------------
# 1. Tokenization (k-mer based)
# ------------------------------

def tokenize_sequence(seq, k=3):
    return [seq[i:i+k] for i in range(len(seq) - k + 1)]

# Example training corpus (use many real sequences in practice)
sequences = [
    "RGDLGDRXGDRGXDMREL",
    "LGDRXGDMRDI",
    "RGXDTRRXGXD",
    "MRELMREIMRDI"
]

k = 3
corpus = [tokenize_sequence(seq, k) for seq in sequences]

# ------------------------------
# 2. Train FastText Model
#    (better than Word2Vec for mutations)
# ------------------------------

model = FastText(
    sentences=corpus,
    vector_size=50,
    window=4,
    min_count=1,
    workers=1,
    sg=1  # skip-gram
)

# ------------------------------
# 3. Combined Similarity Score
#    (Embedding + Edit Distance)
```

```
# ------------------------------

def combined_score(candidate, target="RGD", w_embed=0.7, w_edit=0.3):

    # Embedding similarity
    vec1 = model.wv[candidate].reshape(1, -1)
    vec2 = model.wv[target].reshape(1, -1)
    cosine_sim = cosine_similarity(vec1, vec2)[0][0]

    # Normalized edit similarity
    edit_sim = 1 - (distance(candidate, target) / max(len(candidate), len(target

    # Final weighted score
    final = w_embed * cosine_sim + w_edit * edit_sim
    return final

# ------------------------------
# 4. Evaluate Candidate Motifs
# ------------------------------

target_motif = "RGD"

test_motifs = [
    "RGD",     # highest
    "LGD",     # high
    "DTR",     # lowest
    "RXG",     # middle
    "RGX",     # middle
    "RXGXD"    # low
]

results = {}

for motif in test_motifs:
    results[motif] = combined_score(motif, target_motif)

# Sort by score
sorted_results = sorted(results.items(), key=lambda x: x[1], reverse=True)

print("Ranking relative to target motif:", target_motif)
print("-------------------------------------")

for motif, score in sorted_results:
    print(f"{motif}  -->  {round(score, 4)}")
```

```
Ranking relative to target motif: RGD
-------------------------------------
RGD  -->  1.0
RGX  -->  0.2563999891281128
RXGXD  -->  0.16670000553131104
RXG  -->  0.149399995803833
LGD  -->  0.13019999861717224
DTR  -->  -0.002400000113993883
```

In [44]:

In [45]:
```
from Levenshtein import distance

def find_similar_motifs(sequence, motif, max_mut=1):
    k = len(motif)
```

```
    results = []
    for i in range(len(sequence) - k + 1):
        sub = sequence[i:i+k]
        if distance(sub, motif) <= max_mut:
            results.append((i, sub))
    return results

seq = "LGDMREIMRDLMRDI"
print(find_similar_motifs(seq, "RGD", 1))
print(find_similar_motifs(seq, "MREL", 1))
```

```
[(0, 'LGD')]
[(3, 'MREI'), (7, 'MRDL')]
```

In [45]:

In [46]:
```
import torch
import esm
from sklearn.metrics.pairwise import cosine_similarity

model, alphabet = esm.pretrained.esm2_t6_8M_UR50D()
batch_converter = alphabet.get_batch_converter()

def embed(seq):
    data = [("protein", seq)]
    batch_labels, batch_strs, batch_tokens = batch_converter(data)
    with torch.no_grad():
        results = model(batch_tokens, repr_layers=[6])
    token_repr = results["representations"][6]
    return token_repr.mean(1).numpy()

motif_vec = embed("MREL")
test_vec = embed("MREI")

print(cosine_similarity(motif_vec, test_vec))
```

```
[[0.98645663]]
```

In [46]:

In [47]:
```
from sklearn.cluster import DBSCAN
import numpy as np

k = 4
kmers = [seq[i:i+k] for i in range(len(seq)-k+1)]
embeddings = np.vstack([embed(k) for k in kmers])

clustering = DBSCAN(eps=0.1, min_samples=2).fit(embeddings)

for label in set(clustering.labels_):
    if label != -1:
        cluster = [kmers[i] for i in range(len(kmers)) if clustering.labels_[i]=
        print(cluster)
```

In [47]:

In [48]:
```
# ========================================
# Automatic Motif Discovery via Clustering
# ========================================
```

```
import numpy as np
from gensim.models import FastText
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from collections import defaultdict

# -------------------------
# 1. Tokenization
# -------------------------

def tokenize_sequence(seq, k=3):
    return [seq[i:i+k] for i in range(len(seq)-k+1)]

# Larger synthetic dataset so clustering works
sequences = [
    "RGDLGDRGDRGDLGD",
    "LGDRGDRGXDRGD",
    "RGXDRGDLGD",
    "MRELMREIMRDI",
    "MRDIMRELMREI",
    "DTRDTRDTR",
    "RXGRGDRGXDLGD"
]

k = 3
corpus = [tokenize_sequence(seq, k) for seq in sequences]

# Flatten all k-mers
all_kmers = []
for seq in sequences:
    all_kmers.extend(tokenize_sequence(seq, k))

# -------------------------
# 2. Train FastText
# -------------------------

model = FastText(
    sentences=corpus,
    vector_size=50,
    window=3,
    min_count=1,
    workers=1,
    sg=1
)

# -------------------------
# 3. Create Embedding Matrix
# -------------------------

unique_kmers = list(set(all_kmers))
embeddings = np.array([model.wv[k] for k in unique_kmers])

# Optional dimensionality reduction (helps clustering)
pca = PCA(n_components=10)
reduced = pca.fit_transform(embeddings)

# -------------------------
# 4. Clustering (DBSCAN)
# -------------------------
```

```
cluster_model = DBSCAN(eps=0.5, min_samples=2)
labels = cluster_model.fit_predict(reduced)

# -------------------------
# 5. Display Clusters
# -------------------------

clusters = defaultdict(list)

for kmer, label in zip(unique_kmers, labels):
    if label != -1:  # ignore noise
        clusters[label].append(kmer)

print("Discovered Motif Clusters:\n")

for label, members in clusters.items():
    print(f"Cluster {label}: {members}")
```

```
Discovered Motif Clusters:

Cluster 0: ['GXD', 'MRD', 'XDL', 'EIM', 'GRG', 'REI', 'RGX', 'XGR', 'IMR', 'XDR',
'DRG', 'RDT', 'GDR', 'LGD', 'REL', 'RGD', 'RDI', 'TRD', 'DLG', 'GDL', 'RXG', 'MR
E', 'LMR', 'DTR', 'DIM', 'ELM']
```

In [49]:

```
# Now everything fell into **one dense region**, so DBSCAN made one cluster.
# To get **more clusters**, you must change one of these:


# 1 Reduce `eps` (most important)
# Smaller `eps` → stricter similarity → more clusters.
cluster_model = DBSCAN(eps=0.25, min_samples=2)

# Try:
# * 0.4
# * 0.3
# * 0.25
# * 0.2
# Lower = more separation.

# 2 Increase Embedding Dimension
# Low dimension → vectors collapse together.

vector_size=100  # instead of 50.

# 3 Remove PCA (important)
# PCA can squash differences.
cluster_model = DBSCAN(eps=0.5, min_samples=2)
labels = cluster_model.fit_predict(embeddings)


# 4 Increase k-mer size
# Right now k=3 → too small → many motifs look similar.
k = 4

# That naturally separates:
# * RGD*
# * MREL*
# * DTR*

# 5 Use KMeans Instead (For Forced Multiple Clusters)
```

```
# If your exam requires multiple clusters regardless of density:
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=0)
labels = kmeans.fit_predict(embeddings)

# This **guarantees 3 clusters**.
```

In [50]:

```
# ========================================
# Automatic Motif Discovery via Clustering
# ========================================

import numpy as np
from gensim.models import FastText
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from collections import defaultdict

# -------------------------
# 1. Tokenization
# -------------------------

def tokenize_sequence(seq, k=3):
    return [seq[i:i+k] for i in range(len(seq)-k+1)]

# Larger synthetic dataset so clustering works
sequences = [
    "RGDLGDRGDRGDLGD",
    "LGDRGDRGXDRGD",
    "RGXDRGDLGD",
    "MRELMREIMRDI",
    "MRDIMRELMREI",
    "DTRDTRDTR",
    "RXGRGDRGXDLGD"
]

k = 4
corpus = [tokenize_sequence(seq, k) for seq in sequences]

# Flatten all k-mers
all_kmers = []
for seq in sequences:
    all_kmers.extend(tokenize_sequence(seq, k))

# -------------------------
# 2. Train FastText
# -------------------------

model = FastText(
    sentences=corpus,
    vector_size=100,
    window=3,
    min_count=1,
    workers=1,
    sg=1
)

# -------------------------
# 3. Create Embedding Matrix
# -------------------------
```

```
unique_kmers = list(set(all_kmers))
embeddings = np.array([model.wv[k] for k in unique_kmers])

# Optional dimensionality reduction (helps clustering)
pca = PCA(n_components=10)
reduced = pca.fit_transform(embeddings)

# -------------------------
# 4. Clustering (DBSCAN)
# -------------------------

cluster_model = DBSCAN(eps=0.25, min_samples=2)
labels = cluster_model.fit_predict(embeddings)


# -------------------------
# 5. Display Clusters
# -------------------------

clusters = defaultdict(list)

for kmer, label in zip(unique_kmers, labels):
    if label != -1:  # ignore noise
        clusters[label].append(kmer)

print("Discovered Motif Clusters:\n")

for label, members in clusters.items():
    print(f"Cluster {label}: {members}")
```

```
Discovered Motif Clusters:

Cluster 0: ['RDIM', 'ELMR', 'TRDT', 'RGDL', 'GDLG', 'XDRG', 'GDRG', 'RGDR', 'GXD
R', 'DIMR', 'LMRE', 'RGXD', 'MRDI', 'EIMR', 'IMRD', 'RXGR', 'DRGD', 'MREI', 'XDL
G', 'MREL', 'IMRE', 'GRGD', 'DLGD', 'LGDR', 'DRGX', 'RELM', 'XGRG', 'RDTR', 'GXD
L', 'REIM', 'DTRD']
```

In [51]:

```
# ========================================
# Automatic Motif Discovery via Clustering
# ========================================

import numpy as np
from gensim.models import FastText
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from collections import defaultdict

# -------------------------
# 1. Tokenization
# -------------------------

def tokenize_sequence(seq, k=3):
    return [seq[i:i+k] for i in range(len(seq)-k+1)]

# Larger synthetic dataset so clustering works
sequences = [
    "RGDLGDRGDRGDLGD",
    "LGDRGDRGXDRGD",
    "RGXDRGDLGD",
```

```
    "MRELMREIMRDI",
    "MRDIMRELMREI",
    "DTRDTRDTR",
    "RXGRGDRGXDLGD"
]

k = 4
corpus = [tokenize_sequence(seq, k) for seq in sequences]

# Flatten all k-mers
all_kmers = []
for seq in sequences:
    all_kmers.extend(tokenize_sequence(seq, k))

# -------------------------
# 2. Train FastText
# -------------------------

model = FastText(
    sentences=corpus,
    vector_size=100,
    window=3,
    min_count=1,
    workers=1,
    sg=1
)

# -------------------------
# 3. Create Embedding Matrix
# -------------------------

unique_kmers = list(set(all_kmers))
embeddings = np.array([model.wv[k] for k in unique_kmers])

# Optional dimensionality reduction (helps clustering)
pca = PCA(n_components=10)
reduced = pca.fit_transform(embeddings)

# # -------------------------
# # 4. Clustering (DBSCAN)
# # -------------------------

# cluster_model = DBSCAN(eps=0.25, min_samples=2)
# labels = cluster_model.fit_predict(embeddings)

# -------------------------
# 4. Clustering (KMeans)
# -------------------------

from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(embeddings)


# -------------------------
# 5. Display Clusters
# -------------------------

clusters = defaultdict(list)
```

```
for kmer, label in zip(unique_kmers, labels):
    if label != -1:  # ignore noise
        clusters[label].append(kmer)

print("Discovered Motif Clusters:\n")

for label, members in clusters.items():
    print(f"Cluster {label}: {members}")
```

```
Discovered Motif Clusters:

Cluster 2: ['RDIM', 'ELMR', 'TRDT', 'RGDL', 'GDLG', 'XDRG', 'GDRG', 'RGDR', 'GXD
R', 'DIMR', 'LMRE', 'IMRD', 'DRGD', 'IMRE', 'GRGD', 'LGDR', 'DRGX', 'RELM', 'XGR
G', 'GXDL', 'DTRD']
Cluster 0: ['RGXD', 'MRDI', 'RXGR', 'MREI', 'MREL']
Cluster 1: ['EIMR', 'XDLG', 'DLGD', 'RDTR', 'REIM']
```

In [51]:

In [52]:
```
from transformers import AutoModelForMaskedLM
import torch.nn.functional as F

# 1. You MUST use AutoModelForMaskedLM to get 'logits'
model_name = "facebook/esm2_t6_8M_UR50D"
tokenizer = AutoTokenizer.from_pretrained(model_name)
mlm_model = AutoModelForMaskedLM.from_pretrained(model_name)

def get_mutation_hotspots(sequence):
    inputs = tokenizer(sequence, return_tensors="pt")
    with torch.no_grad():
        outputs = mlm_model(**inputs)
        logits = outputs.logits  # Now this exists!

    # Calculate Entropy (Complexity of the model's guess)
    # Higher entropy = Model is confused = High chance of mutation/variation
    probs = F.softmax(logits, dim=-1)
    entropy = -torch.sum(probs * torch.log(probs + 1e-9), dim=-1)

    return entropy[0].numpy()

# Test it
seq = "MAGRGDVEE"
scores = get_mutation_hotspots(seq)

print("Mutation Probability (Entropy) per position:")
for char, score in zip(seq, scores):
    # Higher score means the LLM thinks this spot is 'flexible'
    print(f"Residue: {char} | Score: {score:.4f}")
```

Loading weights: 100% 107/107 [00:00<00:00, 3232.62it/s]

```
Mutation Probability (Entropy) per position:
Residue: M | Score: 0.0001
Residue: A | Score: 0.1196
Residue: G | Score: 1.6074
Residue: R | Score: 1.0725
Residue: G | Score: 1.4820
Residue: D | Score: 1.0210
Residue: V | Score: 2.0051
Residue: E | Score: 1.4883
Residue: E | Score: 1.3498
```

In [52]:

In [53]:
```python
import torch

def get_sequence_probability(sequence):
    inputs = tokenizer(sequence, return_tensors="pt")
    with torch.no_grad():
        outputs = mlm_model(**inputs, labels=inputs["input_ids"])
        # The 'Loss' in a Language Model is the negative log-likelihood
        loss = outputs.loss
        # We convert loss to a probability-like score (lower loss = higher proba
        return torch.exp(-loss).item()

# Compare the original vs mutation
orig_score = get_sequence_probability("MAGRGDVEE")
mutant_score = get_sequence_probability("MAGLGDVEE")

print(f"Original (RGD) Score: {orig_score:.6f}")
print(f"Mutant (LGD) Score: {mutant_score:.6f}")
# If Mutant Score is close to Original, it's a 'legal' mutation.
```

```
Original (RGD) Score: 0.760574
Mutant (LGD) Score: 0.740986
```

In [53]:

In [54]:
```python
from spacy.pipeline import EntityRuler

# Create a 'Protein NER' (Named Entity Recognition) pipeline
nlp_protein = spacy.blank("en")
ruler = nlp_protein.add_pipe("entity_ruler")

# Define 'Rules' for your motifs
patterns = [
    {"label": "PRIMARY_MOTIF", "pattern": "RGD"},
    {"label": "MUTANT_LEVEL_1", "pattern": "LGD"},
    {"label": "MUTANT_LEVEL_2", "pattern": "MREI"}
]
ruler.add_patterns(patterns)

doc = nlp_protein("MAPLGDVEEEMREI")
for ent in doc.ents:
    print(f"Detected {ent.label_} at position {ent.start}: {ent.text}")
```

In [55]:
```python
import spacy
from spacy.tokens import Span

nlp_protein = spacy.blank("en")
# Add the ruler to the pipeline
```

```
ruler = nlp_protein.add_pipe("entity_ruler")

# Define the dictionary of mutations
patterns = [
    {"label": "PRIMARY_MOTIF", "pattern": "RGD"},
    {"label": "MUTANT_LEVEL_1", "pattern": "LGD"},
    {"label": "MUTANT_LEVEL_2", "pattern": "MREI"}
]
ruler.add_patterns(patterns)

# Test sequence
test_seq = "MAPLGDVEEEMREI"
doc = nlp_protein(test_seq)

print(f"Scanning Sequence: {test_seq}")
if not doc.ents:
    print("No exact matches found. (Note: EntityRuler does exact matching only)"

for ent in doc.ents:
    print(f"Found: {ent.text} | Label: {ent.label_} | Start: {ent.start_char} |
```

```
Scanning Sequence: MAPLGDVEEEMREI
No exact matches found. (Note: EntityRuler does exact matching only)
```

In [55]:

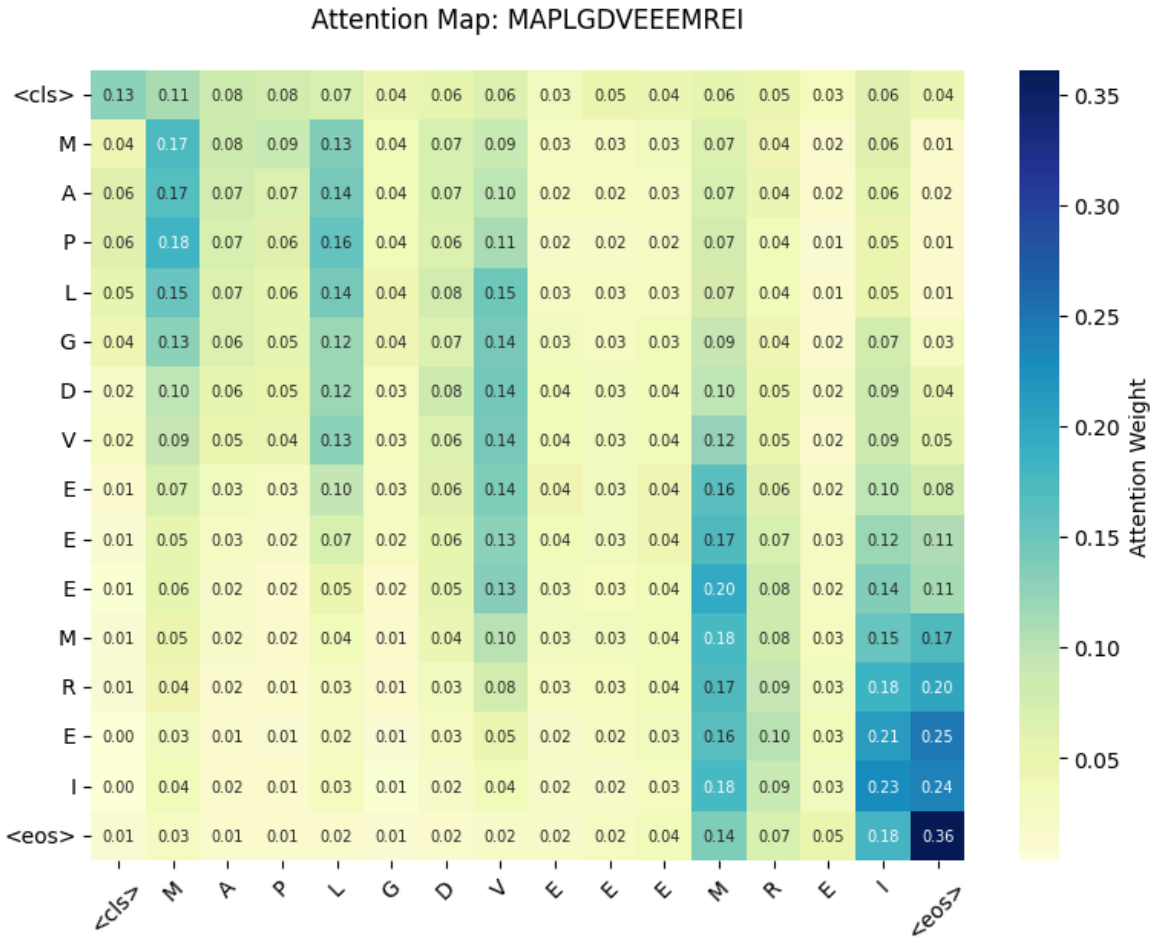
In [56]:
```
import torch
import spacy
import seaborn as sns
import matplotlib.pyplot as plt
import torch.nn.functional as F
from transformers import AutoTokenizer, AutoModelForMaskedLM

# 1. SETUP MODELS
print("Initializing ESM-2 Protein LLM...")
model_name = "facebook/esm2_t6_8M_UR50D"
tokenizer = AutoTokenizer.from_pretrained(model_name)
# We use AutoModelForMaskedLM to get both Logits and Attentions
mlm_model = AutoModelForMaskedLM.from_pretrained(model_name, output_attentions=T

# 2. ZERO-SHOT GRAMMAR CHECKER
def get_sequence_probability(sequence):
    """Calculates how 'biologically natural' a sequence is."""
    inputs = tokenizer(sequence, return_tensors="pt")
    with torch.no_grad():
        outputs = mlm_model(**inputs, labels=inputs["input_ids"])
    # Lower loss means higher probability (exp of negative loss)
    return torch.exp(-outputs.loss).item()

# 3. SPACY MOTIF DICTIONARY
def run_spacy_dictionary(sequence):
    """Uses spaCy to label known motifs and mutants."""
    nlp_protein = spacy.blank("en")
    ruler = nlp_protein.add_pipe("entity_ruler")
    patterns = [
        {"label": "PRIMARY", "pattern": "RGD"},
        {"label": "MUTANT_V1", "pattern": "LGD"},
        {"label": "PRIMARY", "pattern": "MREL"},
        {"label": "MUTANT_V1", "pattern": "MREI"}
    ]
```

```
    ruler.add_patterns(patterns)
    doc = nlp_protein(sequence)
    return [(ent.text, ent.label_, ent.start_char) for ent in doc.ents]

# 4. FIXED ATTENTION VISUALIZATION
def plot_attention(sequence):
    """Visualizes attention with clean, non-overlapping text."""
    inputs = tokenizer(sequence, return_tensors="pt")
    outputs = mlm_model(**inputs)

    # Get last layer attention
    attn = outputs.attentions[-1][0][0].detach().numpy()
    tokens = tokenizer.convert_ids_to_tokens(inputs["input_ids"][0])

    # Calculate dynamic figure size based on sequence length
    fig_size = max(8, len(tokens) * 0.5)
    plt.figure(figsize=(fig_size, fig_size * 0.8))

    # 'annot_kws' controls the text inside the cells
    sns.heatmap(attn,
                xticklabels=tokens,
                yticklabels=tokens,
                annot=True,
                fmt=".2f", # Limit to 2 decimal places
                annot_kws={"size": 7}, # Smaller font for numbers
                cmap="YlGnBu",
                cbar_kws={'label': 'Attention Weight'})

    # Rotate axis labels to prevent overlap
    plt.xticks(rotation=45)
    plt.yticks(rotation=0)

    plt.title(f"Attention Map: {sequence}", pad=20)
    plt.tight_layout() # Adjusts spacing to prevent clipping
    plt.show()



# --- EXECUTION FOR YOUR EXAM ---
test_seq = "MAPLGDVEEEMREI"

print(f"\n--- Analysis for Sequence: {test_seq} ---")

# Part A: spaCy Dictionary
found = run_spacy_dictionary(test_seq)
print(f"1. spaCy Detection: {found}")

# Part B: Grammar Score
prob = get_sequence_probability(test_seq)
print(f"2. Biological Grammar Probability: {prob:.6f}")

# Part C: Visualizing the 'Why'
print("3. Generating Attention Heatmap...")
plot_attention(test_seq)
```

```
Initializing ESM-2 Protein LLM...
```

Loading weights: 100% 107/107 [00:00<00:00, 2170.05it/s]

```
--- Analysis for Sequence: MAPLGDVEEEMREI ---
1. spaCy Detection: []
2. Biological Grammar Probability: 0.684165
3. Generating Attention Heatmap...
```



In [56]:

In [57]:
```python
# ============================================================
# INTEGRATED NLP / LLM-STYLE MOTIF ANALYSIS PIPELINE
# ============================================================

import numpy as np
from gensim.models import FastText
from sklearn.cluster import KMeans
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.ensemble import IsolationForest
from sklearn.metrics import silhouette_score
from Levenshtein import distance
from collections import defaultdict, Counter
import networkx as nx
import random

# ------------------------------------------------------------
# DATA
# ------------------------------------------------------------

sequences = [
    "RGDLGDRGDRGDLGD",
    "LGDRGDRGXDRGD",
    "RGXDRGDLGD",
    "MRELMREIMRDI",
```

```
    "MRDIMRELMREI",
    "DTRDTRDTR",
    "RXGRGDRGXDLGD"
]

target_motif = "RGD"
k = 3

def tokenize_sequence(seq, k=3):
    return [seq[i:i+k] for i in range(len(seq)-k+1)]

corpus = [tokenize_sequence(seq, k) for seq in sequences]
all_kmers = list(set([kmer for seq in sequences for kmer in tokenize_sequence(se

# ------------------------------------------------------------
# 1 --- Biochemical Property Encoding
# ------------------------------------------------------------

print("\n1 --- Biochemical Encoding")

aa_props = {
    'R': [1,0,1], 'K':[1,0,1],   # positive
    'D':[-1,0,1], 'E':[-1,0,1],  # negative
    'G':[0,1,0], 'A':[0,1,0],    # small
    'L':[0,1,1], 'I':[0,1,1],    # hydrophobic
    'M':[0,1,1], 'X':[0,0,0]
}

def encode_motif(motif):
    vec = []
    for aa in motif:
        vec.extend(aa_props.get(aa, [0,0,0]))
    return np.array(vec)

print("Encoded RGD:", encode_motif("RGD"))

# ------------------------------------------------------------
# 2 --- Train FastText Embeddings
# ------------------------------------------------------------

print("\n2 --- FastText Embedding Training")

model = FastText(sentences=corpus,
                 vector_size=50,
                 window=3,
                 min_count=1,
                 workers=1,
                 sg=1)

embeddings = np.array([model.wv[k] for k in all_kmers])
print("Embedding shape:", embeddings.shape)

# ------------------------------------------------------------
# 3 --- N-gram Language Model Probability
# ------------------------------------------------------------

print("\n3 --- N-gram Probability")

all_tokens = [aa for seq in sequences for aa in seq]
bigram_counts = Counter(zip(all_tokens, all_tokens[1:]))
```

```
def bigram_prob(a,b):
    return bigram_counts[(a,b)] / max(1,sum(v for (x,y),v in bigram_counts.items

def motif_probability(motif):
    probs = []
    for i in range(len(motif)-1):
        probs.append(bigram_prob(motif[i], motif[i+1]))
    return np.mean(probs) if probs else 0

print("RGD probability:", motif_probability("RGD"))

# ------------------------------------------------------------
# 4 --- KMeans Clustering
# ------------------------------------------------------------

print("\n4 --- KMeans Clustering")

kmeans = KMeans(n_clusters=3, random_state=42)
labels = kmeans.fit_predict(embeddings)

clusters = defaultdict(list)
for kmer, label in zip(all_kmers, labels):
    clusters[label].append(kmer)

for label, members in clusters.items():
    print(f"Cluster {label}:", members)

# ------------------------------------------------------------
# 5 --- Hybrid Similarity Scoring
# ------------------------------------------------------------

print("\n5 --- Hybrid Scoring")

def hybrid_score(candidate):
    vec1 = model.wv[candidate].reshape(1,-1)
    vec2 = model.wv[target_motif].reshape(1,-1)
    cos = cosine_similarity(vec1, vec2)[0][0]
    edit_sim = 1 - (distance(candidate,target_motif)/max(len(candidate),len(targ
    prob = motif_probability(candidate)
    return 0.5*cos + 0.3*edit_sim + 0.2*prob

candidates = ["RGD","LGD","RXG","RGX","RXD","DTR"]
for c in candidates:
    if c in model.wv:
        print(c, round(hybrid_score(c),4))

# ------------------------------------------------------------
# 6 --- Masked Mutation Prediction (Simple)
# ------------------------------------------------------------

print("\n6 --- Masked Mutation Simulation")

def predict_middle(motif):
    first,last = motif[0], motif[2]
    probs = {aa:bigram_prob(first,aa)+bigram_prob(aa,last) for aa in aa_props.ke
    return sorted(probs.items(), key=lambda x:x[1], reverse=True)[:3]

print("Top middle residue predictions for R?D:", predict_middle("RGD"))
```

```
# ------------------------------------------------------------
# 7 --- Graph-Based Similarity Network
# ------------------------------------------------------------

print("\n7 --- Graph Motif Network")

G = nx.Graph()
for kmer in all_kmers:
    G.add_node(kmer)

for i in range(len(all_kmers)):
    for j in range(i+1,len(all_kmers)):
        sim = cosine_similarity(
            model.wv[all_kmers[i]].reshape(1,-1),
            model.wv[all_kmers[j]].reshape(1,-1)
        )[0][0]
        if sim > 0.8:
            G.add_edge(all_kmers[i], all_kmers[j])

print("Number of connected components:", nx.number_connected_components(G))

# ------------------------------------------------------------
# 8 --- Mutation Simulation + Ranking
# ------------------------------------------------------------

print("\n8 --- In-silico Mutation Ranking")

amino_acids = list(aa_props.keys())

def mutate_once(motif):
    pos = random.randint(0,len(motif)-1)
    new_aa = random.choice(amino_acids)
    return motif[:pos] + new_aa + motif[pos+1:]

mutants = [mutate_once("RGD") for _ in range(10)]
mutants = list(set(mutants))

ranked = sorted(mutants, key=lambda x: hybrid_score(x) if x in model.wv else 0,
print("Top mutated variants:", ranked[:5])

# ------------------------------------------------------------
# 9 --- Anomaly Detection
# ------------------------------------------------------------

print("\n9 --- Anomaly Detection")

iso = IsolationForest(contamination=0.2, random_state=42)
iso.fit(embeddings)
anomaly_scores = iso.predict(embeddings)

for kmer, score in zip(all_kmers, anomaly_scores):
    if score == -1:
        print("Anomalous motif:", kmer)

print("\nPipeline Complete.")
```

```
1 --- Biochemical Encoding
Encoded RGD: [ 1  0  1  0  1  0 -1  0  1]

2 --- FastText Embedding Training
Embedding shape: (26, 50)

3 --- N-gram Probability
RGD probability: 0.625

4 --- KMeans Clustering
Cluster 0: ['GXD', 'XDL', 'IMR', 'XDR', 'RDT', 'GDR', 'REL', 'RGD', 'RDI', 'MRE',
'DTR', 'ELM']
Cluster 2: ['MRD', 'GRG', 'RGX', 'XGR', 'TRD', 'DLG', 'GDL', 'LMR']
Cluster 1: ['EIM', 'REI', 'DRG', 'LGD', 'RXG', 'DIM']

5 --- Hybrid Scoring
RGD 0.925
LGD 0.3355
RXG 0.1477
RGX 0.2854
RXD 0.2453
DTR 0.1089

6 --- Masked Mutation Simulation
Top middle residue predictions for R?D: [('G', 1.25), ('X', 0.8), ('R', 0.25)]

7 --- Graph Motif Network
Number of connected components: 26

8 --- In-silico Mutation Ranking
Top mutated variants: ['MGD', 'RGM', 'KGD', 'RGE', 'LGD']

9 --- Anomaly Detection
Anomalous motif: XDL
Anomalous motif: REI
Anomalous motif: GDR
Anomalous motif: RDI
Anomalous motif: DIM

Pipeline Complete.
```

In [57]: