

```
In [1]: # Cell 1: Import all necessary Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import time
import warnings
warnings.filterwarnings('ignore')

# Load the dataset
print("Loading Nivolumab binding data...")
df = pd.read_csv('Nivolumab_w_zero_conc.csv')
print(f"Dataset shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")
print("\nFirst 5 rows:")
print(df.head())

# Extract concentration columns
sequences = df['Sequence'].values
conc_0 = df['0pM'].values      # Buffer only
conc_63 = df['63pM'].values    # 63 pM
conc_250 = df['250pM'].values  # 250 pM
conc_1000 = df['1000pM'].values # 1000 pM
conc_4000 = df['4000pM'].values # 4000 pM
```

Loading Nivolumab binding data...
Dataset shape: (246450, 6)
Columns: ['Sequence', '0pM', '63pM', '250pM', '1000pM', '4000pM']

First 5 rows:

	Sequence	0pM	63pM	250pM	1000pM	4000pM
0	HRQPKGPHDN	174	178.000000	182.000000	210.333333	309.000000
1	YFEDRNTYMP	169	176.000000	181.000000	203.000000	353.333333
2	ANGPVNWGEN	173	177.333333	179.333333	198.000000	273.333333
3	PELMAPITED	172	173.000000	180.666667	196.333333	273.000000
4	DMASFEDVDY	176	177.333333	177.000000	196.666667	274.333333

```
In [2]: import requests
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
import py3Dmol

# Download PD-1 sequence from UniProt
print("Downloading PD-1 sequence from UniProt (ID: Q15116)...")
uniprot_id = "Q15116"
url = f"https://www.uniprot.org/uniprot/{uniprot_id}.fasta"
response = requests.get(url)
```

```

if response.status_code == 200:
    fasta_lines = response.text.strip().split('\n')
    header = fasta_lines[0]
    pd1_sequence = ''.join(fasta_lines[1:])

    print(f"\nHeader: {header}")
    print(f"\nPD-1 Sequence Length: {len(pd1_sequence)}")
    print(f"\nPD-1 Sequence:\n{pd1_sequence}")

    # Count cysteines
    cysteine_count = pd1_sequence.count('C')
    print(f"\nNumber of cysteines in PD-1: {cysteine_count}")

```

Downloading PD-1 sequence from UniProt (ID: Q15116)...

Header: >sp|Q15116|PDCD1_HUMAN Programmed cell death protein 1 OS=Homo sapiens OX=96
06 GN=PDCD1 PE=1 SV=3

PD-1 Sequence Length: 288

PD-1 Sequence:

MQIPQAPWPVVWAVLQLGWRPGWFLDSPDRPWNPTFSPALLVTEGDNATFTCSFSNTSESFVLNWYRMSPSNQTDKLAAPPE
DRSQPGQDCRFRVTQLPNGRDFHMSVVRARRNDSTYLCGAISLAPKAQIKESLRAELRVTERRAEVPTAHPSPSPRPAGQFQT
LVGVVGGLLGSVLVVWVLAVICSRAARGTIGARRTGQPLKEDPSAVPVFSVDYGELDFQWREKTPEPPVPCVPEQTEYATIV
FPSGMGTSSPARRGSADGPRSAQPLRPEDGHCSWPL

Number of cysteines in PD-1: 6

```

In [3]: #Cell 2 Create overlapping 10-mer tiles
peptide_length = 10
tiles = []
tile_positions = []
original_tiles = []

print("\nCreating overlapping tiles...")
for i in range(len(pd1_sequence) - peptide_length + 1):
    original_tile = pd1_sequence[i:i + peptide_length]
    # Replace cysteine with alanine
    modified_tile = original_tile.replace('C', 'A')

    original_tiles.append(original_tile)
    tiles.append(modified_tile)
    tile_positions.append(i)

print(f"Total number of tiles: {len(tiles)}")
print(f"\nFirst 5 tiles (C->A replacements shown):")
for i in range(min(5, len(tiles))):
    print(f"  Position {tile_positions[i]+1}-{tile_positions[i]+10}: {original_tile}")

```

```
Creating overlapping tiles...
```

```
Total number of tiles: 279
```

```
First 5 tiles (C->A replacements shown):
Position 1-10: MQIPQAPWPV -> MQIPQAPWPV
Position 2-11: QIPQAPWPVV -> QIPQAPWPVV
Position 3-12: IPQAPWPVVW -> IPQAPWPVVW
Position 4-13: PQAPWPVVWA -> PQAPWPVVWA
Position 5-14: QAPWPVVWAV -> QAPWPVVWAV
```

```
In [4]:
```

```
# Cell 3: Load the trained model and create tiles (CORRECTED)
# Define the model architecture (must match your trained model)
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size=190, hidden_size=50, output_size=4, dropout_rate=0.2):
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Load model - CORRECTED to handle the saved format
print("Loading trained model...")
model = NivolumabPredictor()

# Check if the saved file contains a dictionary or just state_dict
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
if isinstance(checkpoint, dict) and 'model_state_dict' in checkpoint:
    # New format with multiple items
    model.load_state_dict(checkpoint['model_state_dict'])
    print("Model loaded from checkpoint format")
else:
    # Old format with just state_dict
    model.load_state_dict(checkpoint)
    print("Model loaded from state_dict format")

model.eval()
print("Model loaded successfully!")

# Amino acid mapping (19 amino acids, no C)
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Create overlapping tiles
peptide_length = 10
tiles = []
tile_positions = []
original_tiles = []

print("\nCreating overlapping tiles...")
```

```

for i in range(len(pd1_sequence) - peptide_length + 1):
    original_tile = pd1_sequence[i:i + peptide_length]
    # Replace cysteine with alanine
    modified_tile = original_tile.replace('C', 'A')

    original_tiles.append(original_tile)
    tiles.append(modified_tile)
    tile_positions.append(i)

print(f"Total number of tiles: {len(tiles)}")
print(f"\nFirst 5 tiles (showing C->A replacements):")
for i in range(min(5, len(tiles))):
    if original_tiles[i] != tiles[i]:
        print(f"  Position {tile_positions[i]+1}-{tile_positions[i]+10}: {original_tiles[i]}")
    else:
        print(f"  Position {tile_positions[i]+1}-{tile_positions[i]+10}: {tiles[i]}")

```

Loading trained model...
Model loaded from state_dict format
Model loaded successfully!

Creating overlapping tiles...
Total number of tiles: 279

First 5 tiles (showing C->A replacements):
Position 1-10: MQIPQAPWPV
Position 2-11: QIPQAPWPVV
Position 3-12: IPQAPWPVWW
Position 4-13: PQAPWPVVWA
Position 5-14: QAPWPVVWAV

In [5]: # Debug: Check what's in the saved model file

```

import torch

# Load and inspect the saved file
saved_data = torch.load('Nivo_model.pth', map_location='cpu')

print("Type of saved data:", type(saved_data))
if isinstance(saved_data, dict):
    print("Keys in saved dictionary:", saved_data.keys())
    if 'model_architecture' in saved_data:
        print("Model architecture:", saved_data['model_architecture'])

```

Type of saved data: <class 'collections.OrderedDict'>
Keys in saved dictionary: odict_keys(['fc1.weight', 'fc1.bias', 'fc2.weight', 'fc2.biases'])

In [6]: # Cell 4: To save (for future reference):

```

torch.save({
    'model_state_dict': model.state_dict(),
    'model_architecture': {
        'input_size': 190,
        'hidden_size': 50,
        'output_size': 4,
        'dropout_rate': 0
    }
}, 'Nivo_model.pth')

```

```
# To Load:
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
model = NivolumabPredictor(**checkpoint['model_architecture'])
model.load_state_dict(checkpoint['model_state_dict'])
model.eval()
```

```
Out[6]: NivolumabPredictor(
    (fc1): Linear(in_features=190, out_features=50, bias=True)
    (relu): ReLU()
    (dropout): Dropout(p=0, inplace=False)
    (fc2): Linear(in_features=50, out_features=4, bias=True)
)
```

```
In [7]: # Cell 5: Load the trained model correctly
import torch
import torch.nn as nn
import numpy as np

# Define the model architecture
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size=190, hidden_size=50, output_size=4, dropout_rate=0.2):
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Load model - handle both old and new save formats
try:
    # Try Loading as a checkpoint with multiple items
    checkpoint = torch.load('Nivo_model.pth', map_location='cpu')

    if isinstance(checkpoint, dict) and 'model_state_dict' in checkpoint:
        # New format with model_state_dict
        model = NivolumabPredictor()
        model.load_state_dict(checkpoint['model_state_dict'])
        print("Model loaded from checkpoint format")
    else:
        # Old format - just state dict
        model = NivolumabPredictor()
        model.load_state_dict(checkpoint)
        print("Model loaded from state dict format")
except:
    # If all else fails, try loading just the state dict
    model = NivolumabPredictor()
    state_dict = torch.load('Nivo_model.pth', map_location='cpu')
    model.load_state_dict(state_dict)
    print("Model loaded directly")
```

```

model.eval()
print("Model loaded successfully and set to evaluation mode")

# Define amino acid mapping (19 amino acids, no C)
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Function to encode sequences
def encode_sequence(sequence):
    """One-hot encode a 10-mer peptide sequence"""
    one_hot = torch.zeros(190) # 10 positions x 19 amino acids
    for i, aa in enumerate(sequence[:10]):
        if aa in aa_to_idx:
            one_hot[i * 19 + aa_to_idx[aa]] = 1
    return one_hot

# Make sure tiles are defined from previous cells
if 'tiles' not in globals():
    print("ERROR: 'tiles' variable not found. Please run the previous cells that cr
else:
    # Predict binding for all tiles
    print(f"\nPredicting binding for {len(tiles)} PD-1 tiles...")
    predictions = []

    with torch.no_grad():
        for i, tile in enumerate(tiles):
            if i % 50 == 0:
                print(f"  Processing tile {i}/{len(tiles)}...")

            one_hot = encode_sequence(tile)
            log_pred = model(one_hot.unsqueeze(0))

            # Get 4000 pM prediction (index 3) and convert from Log10
            log_binding_4000pM = log_pred[0, 3].item()
            linear_binding = 10 ** log_binding_4000pM
            predictions.append(linear_binding)

    predictions = np.array(predictions)

    print(f"\nPrediction complete!")
    print(f"Binding statistics at 4000 pM:")
    print(f"  Min: {predictions.min():.2f}")
    print(f"  Max: {predictions.max():.2f}")
    print(f"  Mean: {predictions.mean():.2f}")
    print(f"  Median: {np.median(predictions):.2f}")

```

```
Model loaded from checkpoint format
Model loaded successfully and set to evaluation mode
```

```
Predicting binding for 279 PD-1 tiles...
Processing tile 0/279...
Processing tile 50/279...
Processing tile 100/279...
Processing tile 150/279...
Processing tile 200/279...
Processing tile 250/279...
```

```
Prediction complete!
Binding statistics at 4000 pM:
Min: 55.49
Max: 130653.29
Mean: 1917.27
Median: 101.86
```

```
In [8]: # Cell 6: to check what's in the saved model file
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
print("Type of loaded object:", type(checkpoint))

if isinstance(checkpoint, dict):
    print("\nKeys in checkpoint:")
    for key in checkpoint.keys():
        print(f" - {key}")
        if key == 'model_architecture':
            print(f"    Architecture: {checkpoint[key]}")
else:
    print("\nLoaded object appears to be a state dict directly")
```

```
Type of loaded object: <class 'dict'>
```

```
Keys in checkpoint:
- model_state_dict
- model_architecture
    Architecture: {'input_size': 190, 'hidden_size': 50, 'output_size': 4, 'dropout_rate': 0}
```

```
In [9]: # Cell 7: Find top 10 binding tiles
top_10_indices = np.argsort(predictions)[-10:][::-1]

print("\nTop 10 binding tiles at 4000 pM:")
print("-" * 80)
print(f"{'Rank':<6}{'Position':<15}{'Modified Seq':<15}{'Original Seq':<15}{'Binding':<15}")
print("-" * 80)

for rank, idx in enumerate(top_10_indices, 1):
    position = tile_positions[idx]
    modified_seq = tiles[idx]
    original_seq = original_tiles[idx]
    binding = predictions[idx]
    print(f"{rank:<6}{position+1}-{position+10:<14}{modified_seq:<15}{original_seq:<15}{binding:<15}")

# Store highest binding tile
highest_binding_idx = top_10_indices[0]
```

```

highest_binding_tile = tiles[highest_binding_idx]
highest_binding_position = tile_positions[highest_binding_idx]
highest_binding_value = predictions[highest_binding_idx]

```

Top 10 binding tiles at 4000 pM:

Rank	Position	Modified Seq	Original Seq	Binding
1	28-37	PDRPWNPPPTF	PDRPWNPPPTF	130653.29
2	23-32	WFLDSPDRPW	WFLDSPDRPW	119287.34
3	24-33	FLDSPDRPWN	FLDSPDRPWN	85909.66
4	22-31	GWFLDSPDRP	GWFLDSPDRP	66171.66
5	25-34	LDSPDRPNP	LDSPDRPNP	42432.81
6	26-35	DSPDRPNPP	DSPDRPNPP	37668.00
7	27-36	SPDRPWNPPT	SPDRPWNPPT	8701.89
8	21-30	PGWFLDSPDR	PGWFLDSPDR	7334.10
9	20-29	RPGWFLDSPD	RPGWFLDSPD	5110.62
10	264-273	RRGSADGPRS	RRGSADGPRS	984.08

```

In [10]: # Cell 8: Create binding map where each AA gets the binding value of the tile where
sequence_length = len(pd1_sequence)
binding_map = np.zeros(sequence_length)

# Assign minimum value to first 4 and last 5 positions
min_binding = predictions.min()
binding_map[:4] = min_binding
binding_map[-5:] = min_binding

# Assign binding values where each position is the 5th residue
for i in range(4, sequence_length - 5):
    tile_idx = i - 4
    if tile_idx < len(predictions):
        binding_map[i] = predictions[tile_idx]

print(f"\nBinding map created for {sequence_length} positions")

```

Binding map created for 288 positions

```

In [11]: # Cell 9: for Heat Map Matrix Visualization
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Check that all required variables exist
if 'pd1_sequence' not in globals() or 'binding_map' not in globals():
    print("ERROR: Required variables not found. Please run previous cells first.")
else:
    # Create heat map matrix with 50 columns per row
    n_cols = 50
    sequence_length = len(pd1_sequence)
    n_rows = int(np.ceil(sequence_length / n_cols))

    print(f"Creating heat map for {sequence_length} amino acids")
    print(f"Matrix dimensions: {n_rows} rows x {n_cols} columns")

    # Create matrices for values and amino acid labels
    matrix = np.full((n_rows, n_cols), np.nan, dtype=np.float32)

```

```

aa_matrix = np.empty((n_rows, n_cols), dtype='U1')
aa_matrix.fill('')

# Fill the matrices
for i in range(sequence_length):
    row = i // n_cols
    col = i % n_cols
    matrix[row, col] = binding_map[i]
    aa_matrix[row, col] = pd1_sequence[i]

# Create the figure
fig, ax = plt.subplots(figsize=(20, n_rows * 0.8))

# Create masked array to handle NaN values
masked_matrix = np.ma.masked_invalid(matrix)

# Plot the heat map
im = ax.imshow(masked_matrix, cmap='YlOrRd', aspect='equal',
                vmin=np.nanmin(matrix), vmax=np.nanmax(matrix),
                interpolation='nearest')

# Add amino acid labels
for i in range(n_rows):
    for j in range(n_cols):
        if not np.isnan(matrix[i, j]):
            # Determine text color based on background intensity
            val = matrix[i, j]
            # Use white text on dark backgrounds, black on light
            threshold = np.nanmin(matrix) + 0.6 * (np.nanmax(matrix) - np.nanmin(matrix))
            text_color = 'white' if val > threshold else 'black'

            # Add text
            ax.text(j, i, aa_matrix[i, j],
                    ha='center', va='center',
                    fontsize=8, fontweight='bold',
                    color=text_color)

# Add position labels on the left
for i in range(n_rows):
    start_pos = i * n_cols + 1
    end_pos = min((i + 1) * n_cols, sequence_length)
    ax.text(-2, i, f"{start_pos}-{end_pos}",
            ha='right', va='center', fontsize=8)

# Add colorbar
cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
cbar.set_label('Predicted Binding at 4000 pM', fontsize=12)

# Set title and labels
plt.title('PD-1 Sequence Heat Map - Predicted Nivolumab Binding at 4000 pM',
          fontsize=16, pad=20)
plt.xlabel('Position in Row (1-50)', fontsize=12)
plt.ylabel('Row Number', fontsize=12)

# Add grid lines
ax.set_xticks(np.arange(n_cols + 1) - 0.5, minor=True)

```

```

        ax.set_yticks(np.arange(n_rows + 1) - 0.5, minor=True)
        ax.grid(which='minor', color='gray', linestyle='-', linewidth=0.5, alpha=0.5)

    # Remove major ticks
    ax.set_xticks([])
    ax.set_yticks([])

    # Highlight the highest binding region if available
    if 'highest_binding_position' in globals():
        # Add rectangles around the highest binding 10-mer
        for offset in range(10):
            pos = highest_binding_position + offset
            if pos < sequence_length:
                row = pos // n_cols
                col = pos % n_cols
                rect = patches.Rectangle((col-0.5, row-0.5), 1, 1,
                                         linewidth=2, edgecolor='blue',
                                         facecolor='none', zorder=10)
                ax.add_patch(rect)

        # Add legend for the highlight
        blue_patch = patches.Patch(color='none', edgecolor='blue', linewidth=2,
                                    label=f'Highest binding region\n({highest_binding}')
        ax.legend(handles=[blue_patch], loc='upper right', bbox_to_anchor=(1.15, 1))

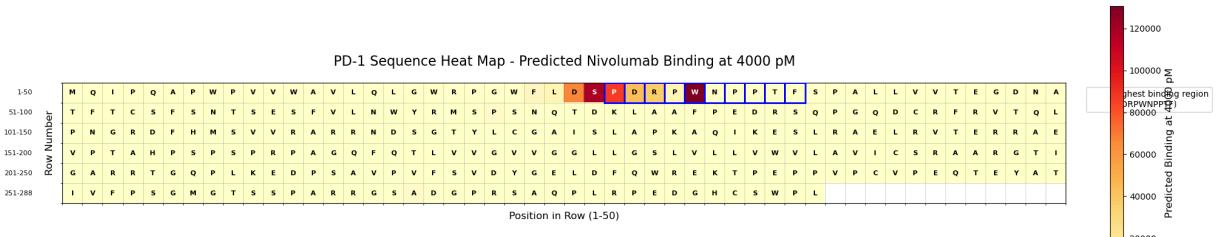
    # Adjust layout
    plt.tight_layout()

    # Save the figure
    plt.savefig('pd1_binding_heatmap_matrix.png', dpi=300, bbox_inches='tight')
    plt.show()

    # Print summary statistics
    print(f"\nHeat map created successfully!")
    print(f"Binding value range: {np.nanmin(matrix):.2f} - {np.nanmax(matrix):.2f}")
    print(f"Number of amino acids displayed: {np.sum(~np.isnan(matrix))}")

```

Creating heat map for 288 amino acids
Matrix dimensions: 6 rows x 50 columns



Heat map created successfully!
Binding value range: 55.49 - 130653.29
Number of amino acids displayed: 288

```
In [12]: # 9': Alternative Simplified Heat Map
import numpy as np
import matplotlib.pyplot as plt

# Create a linear heat map if matrix version has issues
```

```

if 'pd1_sequence' in globals() and 'binding_map' in globals():
    plt.figure(figsize=(20, 4))

    # Create position array
    positions = np.arange(len(pd1_sequence))

    # Normalize binding values for coloring
    norm_binding = (binding_map - binding_map.min()) / (binding_map.max() - binding_map.min())
    colors = plt.cm.YlOrRd(norm_binding)

    # Create bar plot where height is constant but color varies
    bars = plt.bar(positions, np.ones_like(positions), color=colors, width=1.0, edgecolor='black')

    # Add amino acid labels at regular intervals
    label_interval = 10
    for i in range(0, len(pd1_sequence), label_interval):
        plt.text(i, 0.5, pd1_sequence[i], ha='center', va='center', fontsize=8, rotation=0)
        plt.text(i, -0.1, str(i+1), ha='center', va='top', fontsize=6)

    # Highlight highest binding region if available
    if 'highest_binding_position' in globals():
        plt.axvspan(highest_binding_position, highest_binding_position + 10, alpha=0.3, color='blue', ymin=0, ymax=1)
        plt.text(highest_binding_position + 5, 1.1, f'Highest\nBinding\n{highest_binding_tile}', ha='center', va='bottom', fontsize=10, color='blue')

    # Add known binding regions from crystal structure
    plt.axvspan(24, 35, alpha=0.2, color='red', ymin=0.8, ymax=1)
    plt.text(29.5, 1.2, 'N-loop', ha='center', fontsize=8, color='red')

    plt.axvspan(44, 55, alpha=0.2, color='green', ymin=0.8, ymax=1)
    plt.text(49.5, 1.2, 'BC-loop', ha='center', fontsize=8, color='green')

    plt.axvspan(124, 135, alpha=0.2, color='orange', ymin=0.8, ymax=1)
    plt.text(129.5, 1.2, 'FG-loop', ha='center', fontsize=8, color='orange')

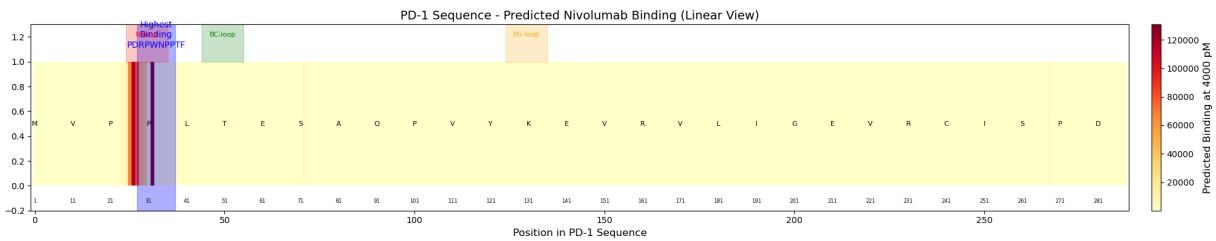
    plt.xlim(-1, len(pd1_sequence))
    plt.ylim(-0.2, 1.3)
    plt.xlabel('Position in PD-1 Sequence', fontsize=12)
    plt.title('PD-1 Sequence - Predicted Nivolumab Binding (Linear View)', fontsize=14)

    # Add colorbar
    sm = plt.cm.ScalarMappable(cmap=plt.cm.YlOrRd,
                                 norm=plt.Normalize(vmin=binding_map.min(),
                                                    vmax=binding_map.max()))
    sm.set_array([])
    cbar = plt.colorbar(sm, ax=plt.gca(), fraction=0.02, pad=0.02)
    cbar.set_label('Predicted Binding at 4000 pM', fontsize=12)

    plt.tight_layout()
    plt.savefig('pd1_binding_linear_heatmap.png', dpi=300, bbox_inches='tight')
    plt.show()

print("Linear heat map created successfully!")

```



Linear heat map created successfully!

```
In [13]: # Cell 10: Create binding profile plot with crystal structure annotations
plt.figure(figsize=(16, 8))

positions = np.arange(len(binding_map))
plt.plot(positions + 1, binding_map, 'b-', linewidth=2, label='Predicted Binding')

# Known binding regions from crystal structure
plt.axvspan(25, 35, alpha=0.3, color='red', label='N-loop (Crystal)')
plt.axvspan(45, 55, alpha=0.3, color='green', label='BC-loop (Crystal)')
plt.axvspan(125, 135, alpha=0.3, color='orange', label='FG-loop (Crystal)')

# Make sure highest_binding_tile is defined before using it
if 'highest_binding_tile' not in locals():
    # If not defined, get it from the tiles
    highest_binding_idx = top_10_indices[0]
    highest_binding_tile = tiles[highest_binding_idx]
    highest_binding_position = tile_positions[highest_binding_idx]

# Highlight highest binding region
plt.axvspan(highest_binding_position + 1, highest_binding_position + 10,
            alpha=0.5, color='purple',
            label=f'Highest Predicted\nn({highest_binding_tile})')

plt.xlabel('Amino Acid Position', fontsize=14)
plt.ylabel('Predicted Binding at 4000 pM', fontsize=14)
plt.title('PD-1 Binding Profile - Model Predictions vs Crystal Structure', fontsize=16)
plt.legend(loc='best', fontsize=10)
plt.grid(True, alpha=0.3)
plt.xlim(1, len(binding_map))

# Add some additional improvements
plt.ylim(bottom=0) # Start y-axis at 0 for better visualization

# Add text annotation for the highest binding region
plt.text(highest_binding_position + 5, binding_map[highest_binding_position + 4] +
         f'{highest_binding_tile}\n{highest_binding_value:.1f}',
         ha='center', va='bottom', fontsize=10,
         bbox=dict(boxstyle='round,pad=0.3', facecolor='yellow', alpha=0.7))

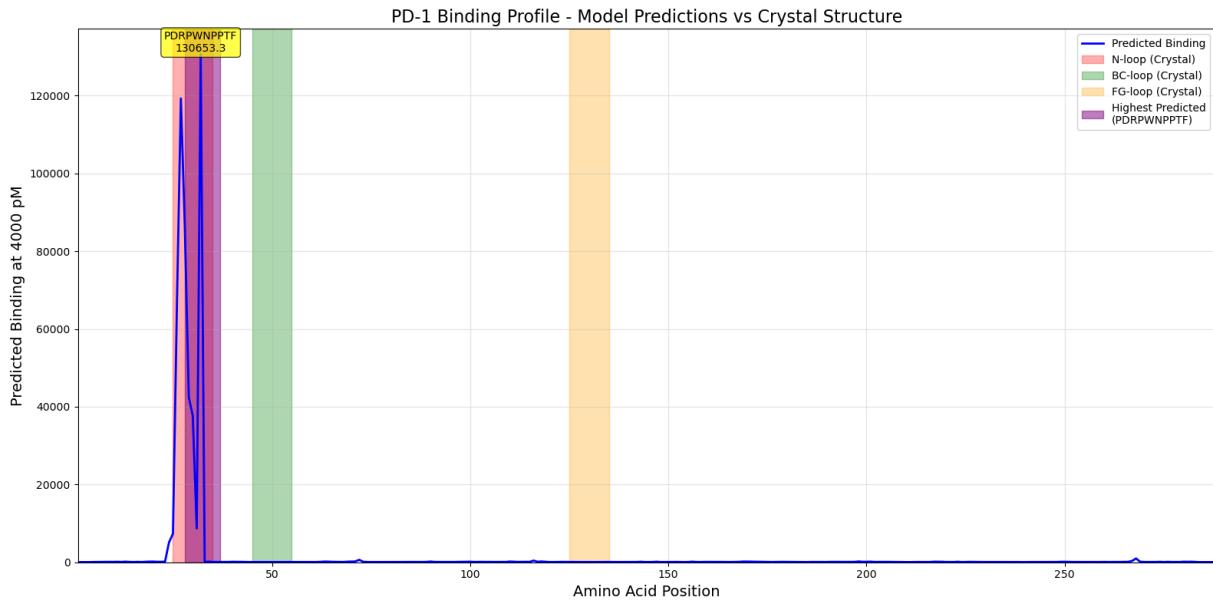
plt.tight_layout()
plt.savefig('pd1_binding_profile.png', dpi=300, bbox_inches='tight')
plt.show()

print(f'\nPredicted epitope region: Position {highest_binding_position+1}-{highest_
print(f"Sequence: {highest_binding_tile}")
print(f"Binding value: {highest_binding_value:.2f}")
```

```

# Additional analysis
print(f"\nOverlap with known binding sites:")
print(f" N-loop (25-35): {'Yes' if 25 <= highest_binding_position + 5 <= 35 else 'No'}")
print(f" BC-loop (45-55): {'Yes' if 45 <= highest_binding_position + 5 <= 55 else 'No'}")
print(f" FG-loop (125-135): {'Yes' if 125 <= highest_binding_position + 5 <= 135 else 'No'}")

```



Predicted epitope region: Position 28-37

Sequence: PDRPWNPPTF

Binding value: 130653.29

Overlap with known binding sites:

N-loop (25-35): Yes
 BC-loop (45-55): No
 FG-loop (125-135): No

```

In [14]: # Cell 11: Download and visualize PD-1 structure with py3Dmol (FIXED)
import py3Dmol
import requests
import numpy as np

# Download AlphaFold structure
print("Downloading PD-1 AlphaFold structure...")
uniprot_id = "Q15116"
af_url = f"https://alphafold.ebi.ac.uk/files/AF-{uniprot_id}-F1-model_v4.pdb"

try:
    response = requests.get(af_url, timeout=30)
    response.raise_for_status() # Raise an error for bad status codes

    if response.status_code == 200:
        pdb_data = response.text
        print("AlphaFold structure downloaded successfully!")
        print(f"Structure size: {len(pdb_data)} characters")

        # Verify it's a valid PDB file
        if not pdb_data.startswith('ATOM') and 'ATOM' not in pdb_data[:1000]:
            print("Warning: Downloaded file may not be a valid PDB format")
    else:

```

```

        print(f"Failed to download structure. Status code: {response.status_code}")
        raise Exception("Download failed")

except requests.exceptions.RequestException as e:
    print(f"Error downloading structure: {e}")
    print("Trying alternative download method...")

    # Alternative: Try downloading with different headers
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
    }
    response = requests.get(af_url, headers=headers, timeout=30)

    if response.status_code == 200:
        pdb_data = response.text
        print("Structure downloaded successfully with alternative method!")
    else:
        raise Exception("Could not download AlphaFold structure")

# Create py3Dmol viewer
print("\nCreating 3D visualization...")
view = py3Dmol.view(width=800, height=600)
view.addModel(pdb_data, 'pdb')

# Normalize binding values for coloring (0-1 range)
norm_binding = (binding_map - binding_map.min()) / (binding_map.max() - binding_map

# Color gradient function
def get_color(value):
    """Convert normalized value (0-1) to color"""
    if value < 0.2:
        return '#E0E0E0' # Light grey for low binding
    elif value < 0.4:
        return '#FFE0B2' # Light orange
    elif value < 0.6:
        return '#FF9800' # Orange
    elif value < 0.8:
        return '#FF5722' # Deep orange
    else:
        return '#D32F2F' # Red for high binding

# Color each residue based on binding value
print("Coloring structure by predicted binding...")
for i in range(sequence_length):
    color = get_color(norm_binding[i])
    # Note: PDB residue numbering starts at 1
    view.setStyle({'chain': 'A', 'resi': i+1},
                  {'cartoon': {'color': color}})

# Highlight the highest binding region with sticks
print(f"Highlighting highest binding region: {highest_binding_tile}")
for i in range(highest_binding_position, min(highest_binding_position + 10, sequence
    view.setStyle({'chain': 'A', 'resi': i+1},
                  {'stick': {'color': 'blue', 'radius': 0.3}}))

# Add Labels for known binding regions from crystal structure

```

```

view.addLabel("N-loop",
              {'fontColor': 'red', 'backgroundColor': 'white', 'backgroundOpacity':
              {'chain': 'A', 'resi': 30})
view.addLabel("BC-loop",
              {'fontColor': 'green', 'backgroundColor': 'white', 'backgroundOpacity':
              {'chain': 'A', 'resi': 50})
view.addLabel("FG-loop",
              {'fontColor': 'orange', 'backgroundColor': 'white', 'backgroundOpacit
              {'chain': 'A', 'resi': 130})

# Add Label for highest binding region
view.addLabel(f"Predicted: {highest_binding_tile}",
              {'fontColor': 'blue', 'backgroundColor': 'white', 'backgroundOpacity':
              {'chain': 'A', 'resi': highest_binding_position + 5})

# Set view and render
view.zoomTo()
view.spin(True)

# Set background color
view.setBackgroundColor('white')

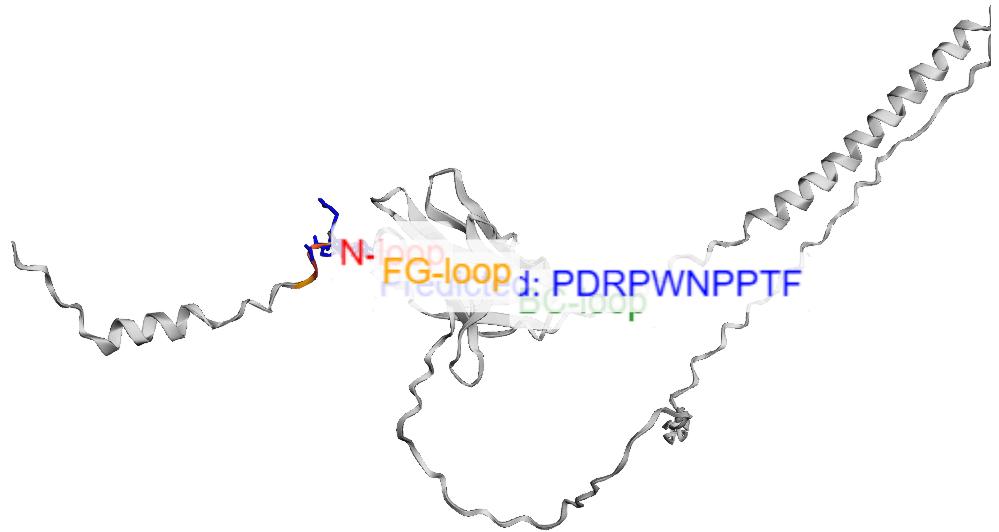
# Display the viewer
view.show()

print("\n3D Structure Visualization Legend:")
print("- Light grey: Low predicted binding")
print("- Orange to Red: Increasing predicted binding")
print("- Blue sticks: Highest binding region")
print("- Labels: Known binding loops from crystal structure")
print("\nUse mouse to rotate, scroll to zoom")

```

Downloading PD-1 AlphaFold structure...
AlphaFold structure downloaded successfully!
Structure size: 186623 characters
Warning: Downloaded file may not be a valid PDB format

Creating 3D visualization...
Coloring structure by predicted binding...
Highlighting highest binding region: PDRPWNPPPTF



3D Structure Visualization Legend:

- Light grey: Low predicted binding
- Orange to Red: Increasing predicted binding
- Blue sticks: Highest binding region
- Labels: Known binding loops from crystal structure

Use mouse to rotate, scroll to zoom

```
In [15]: # Cell 12: Fine mapping - analyze all single amino acid substitutions
print(f"\nFine mapping analysis for highest binding tile: {highest_binding_tile}")

# Create substitution matrix
substitution_matrix = np.zeros((19, 10))

# Original binding for reference
original_one_hot = encode_sequence(highest_binding_tile)
with torch.no_grad():
    original_log_pred = model(original_one_hot.unsqueeze(0))
    original_binding = 10 ** original_log_pred[0, 3].item()
```

```

print(f"Original binding value: {original_binding:.2f}")

# Test all substitutions
for aa_idx, new_aa in enumerate(amino_acids):
    for pos in range(10):
        mutated_seq = list(highest_binding_tile)
        mutated_seq[pos] = new_aa
        mutated_seq = ''.join(mutated_seq)

        one_hot = encode_sequence(mutated_seq)
        with torch.no_grad():
            log_pred = model(one_hot.unsqueeze(0))
            binding = 10 ** log_pred[0, 3].item()

        substitution_matrix[aa_idx, pos] = binding

# Create heat map with improved color scheme
plt.figure(figsize=(12, 10))

# Use a diverging colormap centered on the original binding value
# This makes it easy to see which substitutions improve or worsen binding
from matplotlib.colors import TwoSlopeNorm

# Calculate fold changes for color mapping
fold_change_matrix = substitution_matrix / original_binding

# Create custom colormap - blue for decreased binding, white for no change, red for
norm = TwoSlopeNorm(vmin=0, vcenter=1.0, vmax=fold_change_matrix.max())
im = plt.imshow(fold_change_matrix, cmap='RdBu_r', aspect='auto', norm=norm)

# Add colorbar with clear labels
cbar = plt.colorbar(im, fraction=0.046, pad=0.04)
cbar.set_label('Fold Change vs Wild-Type', fontsize=12)

# Add text annotations showing actual fold changes
for i in range(19):
    for j in range(10):
        fold_change = fold_change_matrix[i, j]
        # Choose text color based on background
        text_color = 'white' if abs(fold_change - 1.0) > 0.5 else 'black'
        plt.text(j, i, f'{fold_change:.1f}', ha='center', va='center',
                 fontsize=8, color=text_color, weight='bold')

plt.yticks(range(19), list(amino_acids))
plt.xticks(range(10), list(highest_binding_tile))
plt.xlabel('Position in Epitope', fontsize=14)
plt.ylabel('Substituted Amino Acid', fontsize=14)
plt.title(f'Single AA Substitution Analysis\nHighest Binding Tile: {highest_binding_tile}')
plt.title(f'Single AA Substitution Analysis\nHighest Binding Tile: {highest_binding_tile}', fontsize=16)

# Highlight wild-type amino acids with thick black border
for pos in range(10):
    wt_aa = highest_binding_tile[pos]
    if wt_aa in amino_acids:
        aa_idx = amino_acids.index(wt_aa)

```

```

        rect = plt.Rectangle((pos-0.45, aa_idx-0.45), 0.9, 0.9,
                             fill=False, edgecolor='black', linewidth=4)
        plt.gca().add_patch(rect)

# Add grid for better readability
plt.grid(True, which='both', color='gray', linewidth=0.5, alpha=0.3)

plt.tight_layout()
plt.savefig('epitope_substitution_heatmap.png', dpi=300, bbox_inches='tight')
plt.show()

# Calculate mean fold-change by position (excluding wild-type)
mean_fold_changes = []
for pos in range(10):
    wt_aa = highest_binding_tile[pos]
    fold_changes_at_pos = []

    for aa_idx, aa in enumerate(amino_acids):
        if aa != wt_aa: # Exclude wild-type
            fold_changes_at_pos.append(fold_change_matrix[aa_idx, pos])

    mean_fold_changes.append(np.mean(fold_changes_at_pos))

# Create bar plot with color coding
plt.figure(figsize=(12, 6))
positions = np.arange(10)

# Color bars based on importance (Lower fold-change = more important)
colors = ['darkred' if fc < 0.5 else 'steelblue' for fc in mean_fold_changes]
bars = plt.bar(positions, mean_fold_changes, color=colors, edgecolor='black', linewidth=1)

# Add value Labels on bars
for i, (bar, fc) in enumerate(zip(bars, mean_fold_changes)):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.02,
             f'{fc:.2f}', ha='center', va='bottom', fontweight='bold')

plt.xticks(positions, list(highest_binding_tile))
plt.xlabel('Position (Wild-type Amino Acid)', fontsize=14)
plt.ylabel('Mean Fold-Change vs Wild-type', fontsize=14)
plt.title('Mean Effect of Substitutions at Each Position\n(Lower values indicate more change)')

# Add reference line at 1.0
plt.axhline(y=1.0, color='red', linestyle='--', alpha=0.5, label='No change', linewidth=1)

# Add threshold line
plt.axhline(y=0.5, color='darkred', linestyle=':', alpha=0.5, label='Critical threshold', linewidth=1)

plt.legend(loc='upper right')
plt.grid(True, axis='y', alpha=0.3)
plt.ylim(0, max(mean_fold_changes) * 1.1)

plt.tight_layout()
plt.savefig('mean_fold_change_by_position.png', dpi=300, bbox_inches='tight')
plt.show()

```

```

# Identify and report the most important positions
sorted_positions = np.argsort(mean_fold_changes)
print("\n" + "*60")
print("EPITOPE FINE MAPPING RESULTS")
print("*60")
print(f"\nOriginal epitope: {highest_binding_tile}")
print(f"Original binding: {original_binding:.2f}")

print("\nThree most critical positions (lowest mean fold-change):")
for i in range(3):
    pos = sorted_positions[i]
    aa = highest_binding_tile[pos]
    print(f" Position {pos+1}: {aa} (mean fold-change: {mean_fold_changes[pos]:.3f}")

    # Find best and worst substitutions at this position
    best_sub_idx = np.argmax(fold_change_matrix[:, pos])
    worst_sub_idx = np.argmin(fold_change_matrix[:, pos])

    print(f"    Best substitution: {amino_acids[best_sub_idx]} ({fold_change_matrix[best_sub_idx, pos]:.3f}")
    print(f"    Worst substitution: {amino_acids[worst_sub_idx]} ({fold_change_matrix[worst_sub_idx, pos]:.3f}")

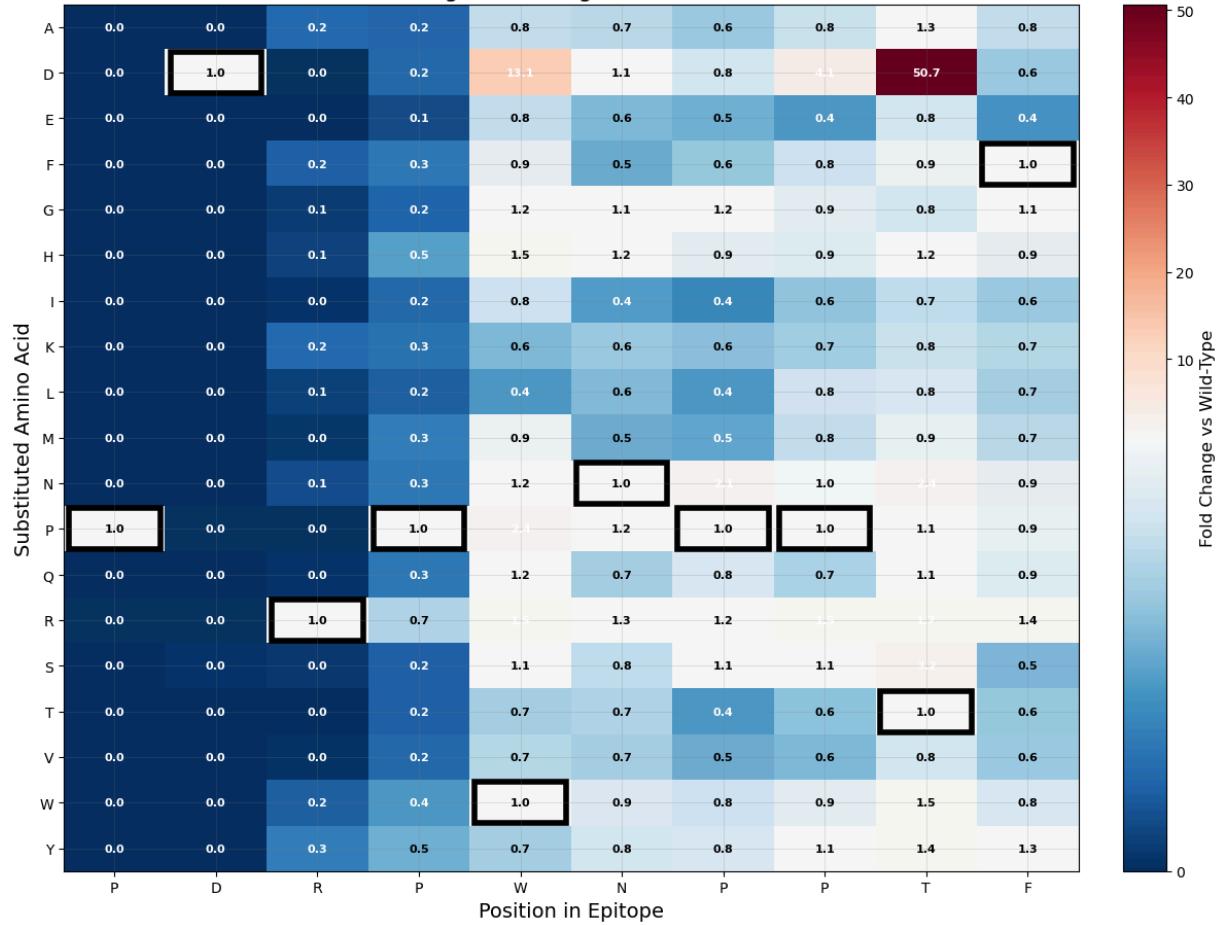
# Find any substitutions that improve binding
improvements = []
for aa_idx, new_aa in enumerate(amino_acids):
    for pos in range(10):
        if fold_change_matrix[aa_idx, pos] > 1.2: # 20% improvement
            wt_aa = highest_binding_tile[pos]
            if new_aa != wt_aa:
                improvements.append((pos+1, wt_aa, new_aa,
                                     substitution_matrix[aa_idx, pos],
                                     fold_change_matrix[aa_idx, pos]))

if improvements:
    print("\nSubstitutions that improve binding (>20% increase):")
    improvements.sort(key=lambda x: x[4], reverse=True) # Sort by fold change
    for pos, wt, new, binding, fold in improvements[:5]:
        print(f" Position {pos}: {wt} → {new}, binding: {binding:.2f} ({fold:.2f}x")

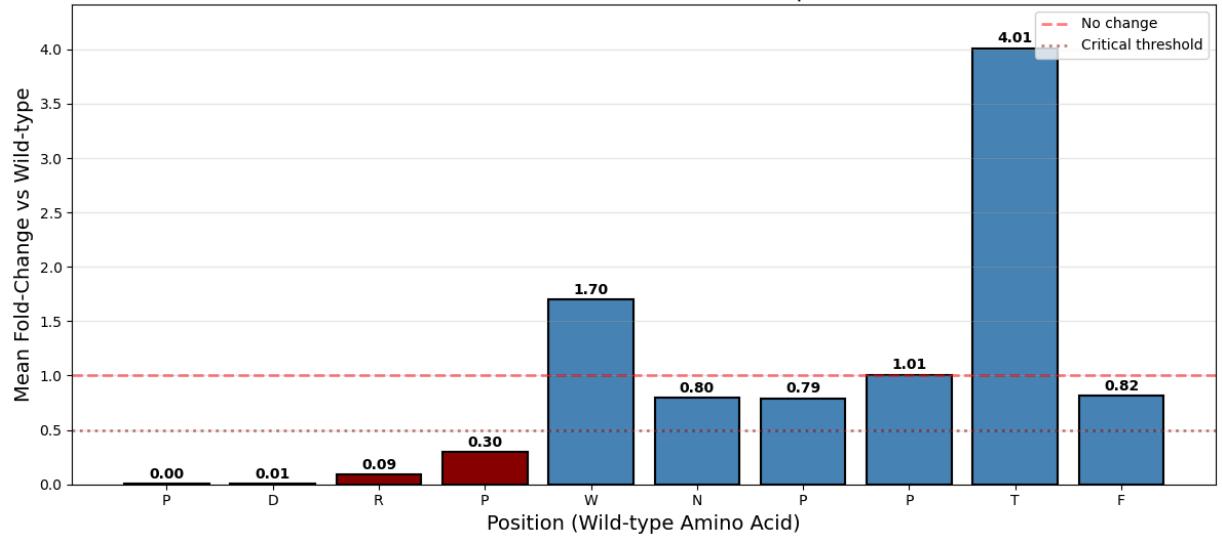
```

Fine mapping analysis for highest binding tile: PDRPWNPPTF
Original binding value: 130653.29

Single AA Substitution Analysis
Highest Binding Tile: PDRPWNPPTF
(Original binding: 130653.29)



Mean Effect of Substitutions at Each Position
(Lower values indicate more critical positions)



```
=====
EPITOPE FINE MAPPING RESULTS
=====
```

Original epitope: PDRPWNPPTF
Original binding: 130653.29

Three most critical positions (lowest mean fold-change):

```
Position 1: P (mean fold-change: 0.003)
    Best substitution: P (1.00x)
    Worst substitution: E (0.00x)
Position 2: D (mean fold-change: 0.005)
    Best substitution: D (1.00x)
    Worst substitution: K (0.00x)
Position 3: R (mean fold-change: 0.094)
    Best substitution: R (1.00x)
    Worst substitution: T (0.01x)
```

Substitutions that improve binding (>20% increase):

```
Position 9: T → D, binding: 6623344.53 (50.69x)
Position 5: W → D, binding: 1712152.49 (13.10x)
Position 8: P → D, binding: 536817.82 (4.11x)
Position 9: T → S, binding: 413939.32 (3.17x)
Position 5: W → P, binding: 312996.63 (2.40x)
```

```
In [16]: # Cell 3: Scatter plot with high contrast and dual regression
plt.figure(figsize=(12, 10))

# Plot all points with high contrast
plt.scatter(conc_1000, conc_4000, alpha=0.6, s=5, c='darkblue',
            edgecolors='none', label='All data points')

# First Linear regression - all points
lr_all = LinearRegression()
X_all = conc_1000.reshape(-1, 1)
lr_all.fit(X_all, conc_4000)
y_pred_all = lr_all.predict(X_all)
r2_all = r2_score(conc_4000, y_pred_all)

# Plot regression line for all points
x_range = np.linspace(conc_1000.min(), conc_1000.max(), 100)
y_range_all = lr_all.predict(x_range.reshape(-1, 1))
plt.plot(x_range, y_range_all, 'r-', linewidth=3,
          label=f'All points: y = {lr_all.coef_[0]:.3f}x + {lr_all.intercept_:.1f}')

# Second regression - excluding saturated points
saturation_level = 65535 # 2^16 - 1
mask_unsaturated = conc_4000 < saturation_level
conc_1000_unsaturated = conc_1000[mask_unsaturated]
conc_4000_unsaturated = conc_4000[mask_unsaturated]

print(f"\nTotal points: {len(conc_4000)}")
print(f"Saturated points: {np.sum(~mask_unsaturated)}")
print(f"Unsaturated points: {np.sum(mask_unsaturated)}")

lr_unsaturated = LinearRegression()
```

```

X_unsaturated = conc_1000_unsaturated.reshape(-1, 1)
lr_unsaturated.fit(X_unsaturated, conc_4000_unsaturated)
y_pred_unsaturated = lr_unsaturated.predict(X_unsaturated)
r2_unsaturated = r2_score(conc_4000_unsaturated, y_pred_unsaturated)

# Plot regression line for unsaturated points
x_range_unsat = np.linspace(conc_1000_unsaturated.min(),
                             conc_1000_unsaturated.max(), 100)
y_range_unsat = lr_unsaturated.predict(x_range_unsat.reshape(-1, 1))
plt.plot(x_range_unsat, y_range_unsat, 'g-', linewidth=3,
          label=f'Unsaturated: y = {lr_unsaturated.coef_[0]:.3f}x + {lr_unsaturated.intercept_:.3f}')

# Add saturation line and highlight saturated points
plt.axhline(y=saturation_level, color='orange', linestyle='--', linewidth=2,
            label=f'Saturation level: {saturation_level:.0f}')
plt.scatter(conc_1000[~mask_unsaturated], conc_4000[~mask_unsaturated],
            alpha=0.8, s=10, c='red', edgecolors='none', label='Saturated points')

plt.xlabel('1000 pM Nivolumab Binding', fontsize=14)
plt.ylabel('4000 pM Nivolumab Binding', fontsize=14)
plt.title('Nivolumab Binding: 4000 pM vs 1000 pM', fontsize=16)
plt.legend(fontsize=10)
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

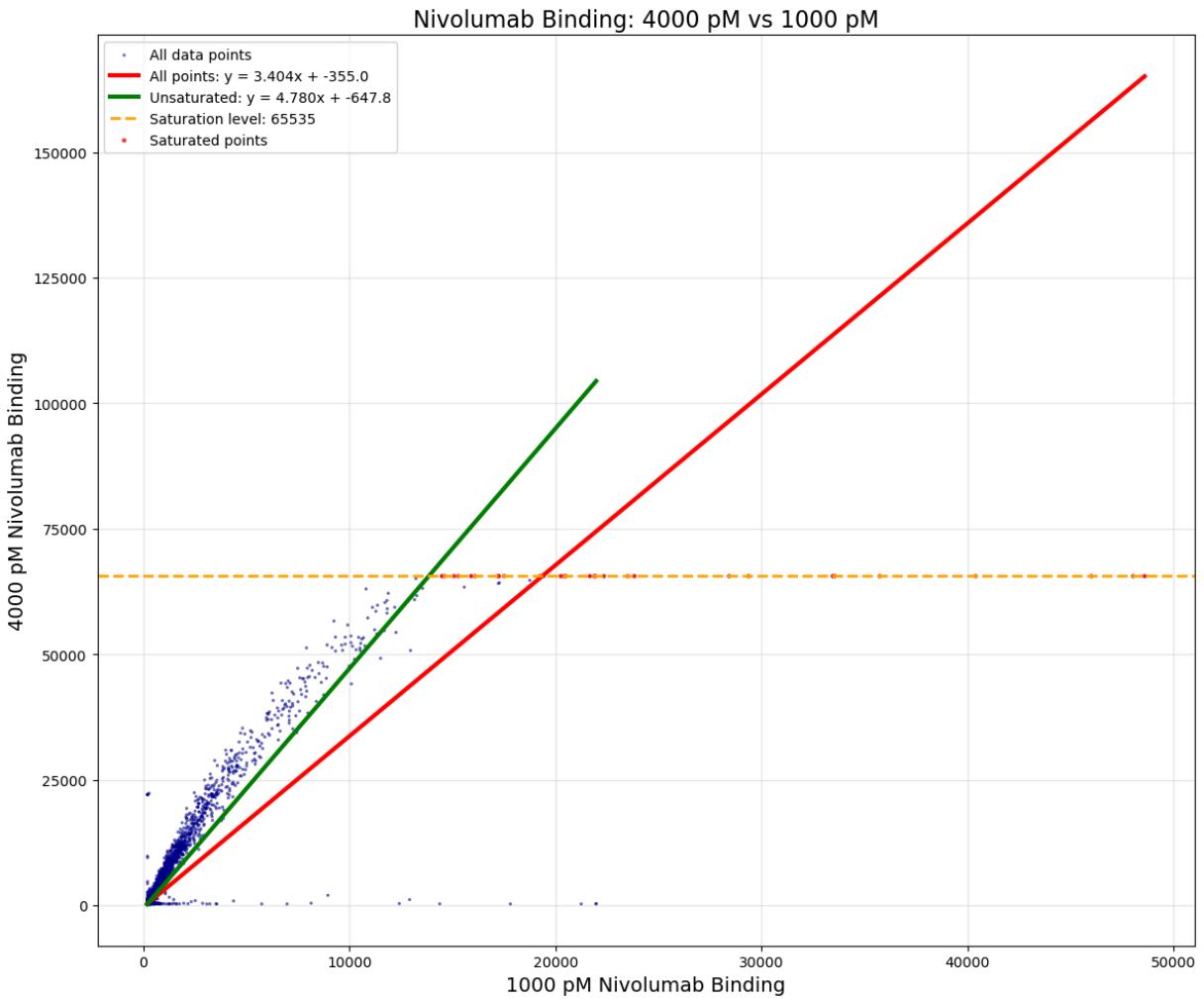
print("\n==== Linear Regression Results ===")
print("\nAll points:")
print(f"Slope: {lr_all.coef_[0]:.4f}")
print(f"Intercept: {lr_all.intercept_:.4f}")
print(f"R-squared: {r2_all:.4f}")
print("\nExcluding saturated points:")
print(f"Slope: {lr_unsaturated.coef_[0]:.4f}")
print(f"Intercept: {lr_unsaturated.intercept_:.4f}")
print(f"R-squared: {r2_unsaturated:.4f}")

```

Total points: 246450

Saturated points: 29

Unsaturated points: 246421



==== Linear Regression Results ===

All points:
 Slope: 3.4042
 Intercept: -355.0280
 R-squared: 0.7272

Excluding saturated points:
 Slope: 4.7802
 Intercept: -647.7596
 R-squared: 0.8184

In [17]: `# Complete Neural Network Implementation for Nivolumab Binding Prediction`

```

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
import time
import warnings
warnings.filterwarnings('ignore')

```

```

# Check for GPU availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")

# Set random seeds for reproducibility
torch.manual_seed(42)
np.random.seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed(42)

# Define parameters (easy to modify)
PARAMS = {
    'hidden_size': 50,
    'learning_rate': 0.001,
    'epochs': 100,
    'batch_size': 128,
    'dropout_rate': 0,
    'weight_decay': 1e-5,
    'train_split': 0.8,
    'val_split': 0.1,
    'test_split': 0.1,
    'shuffle': True,
}

print("\nModel Parameters:")
for key, value in PARAMS.items():
    print(f"  {key}: {value}")

# Load the dataset
print("\nLoading Nivolumab binding data...")
df = pd.read_csv('Nivolumab_w_zero_conc.csv')
print(f"Dataset shape: {df.shape}")
print(f"Columns: {df.columns.tolist()}")

# Extract sequences and concentrations
sequences = df['Sequence'].values
conc_0 = df['0pM'].values      # Buffer only
conc_63 = df['63pM'].values    # 63 pM
conc_250 = df['250pM'].values  # 250 pM
conc_1000 = df['1000pM'].values # 1000 pM
conc_4000 = df['4000pM'].values # 4000 pM

# Determine peptide length
PEPTIDE_LENGTH = len(sequences[0])
print(f"\nPeptide length: {PEPTIDE_LENGTH}")

# Create amino acid mapping (19 amino acids, no C)
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Custom Dataset class
class PeptideDataset(Dataset):
    def __init__(self, sequences, binding_values, peptide_length):
        self.sequences = sequences

```



```

# Define the neural network model
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size, hidden_size=50, output_size=4, dropout_rate=0):
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Initialize model
input_size = PEPTIDE_LENGTH * 19
model = NivolumabPredictor(
    input_size=input_size,
    hidden_size=PARAMS['hidden_size'],
    output_size=4,
    dropout_rate=PARAMS['dropout_rate']
).to(device)

print(f"\nModel architecture:")
print(model)

# Define loss function and optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(),
                       lr=PARAMS['learning_rate'],
                       weight_decay=PARAMS['weight_decay'])

# Training functions
def train_epoch(model, loader, criterion, optimizer):
    model.train()
    total_loss = 0
    for batch_x, batch_y in loader:
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)

        optimizer.zero_grad()
        outputs = model(batch_x)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * batch_x.size(0)

    return total_loss / len(loader.dataset)

def evaluate(model, loader, criterion):
    model.eval()
    total_loss = 0
    all_predictions = []
    all_targets = []

```

```

    with torch.no_grad():
        for batch_x, batch_y in loader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)
            total_loss += loss.item() * batch_x.size(0)

            all_predictions.append(outputs.cpu().numpy())
            all_targets.append(batch_y.cpu().numpy())

    all_predictions = np.concatenate(all_predictions, axis=0)
    all_targets = np.concatenate(all_targets, axis=0)

    return total_loss / len(loader.dataset), all_predictions, all_targets

# Training Loop
print("\n" + "="*60)
print("STARTING TRAINING")
print("=*60")

train_losses = []
val_losses = []
test_losses = []

start_time = time.time()

for epoch in range(PARAMS['epochs']):
    # Train
    train_loss = train_epoch(model, train_loader, criterion, optimizer)

    # Evaluate on validation and test sets
    val_loss, _, _ = evaluate(model, val_loader, criterion)
    test_loss, _, _ = evaluate(model, test_loader, criterion)

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    test_losses.append(test_loss)

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0 or epoch == 0:
        elapsed = time.time() - start_time
        print(f"Epoch [{epoch+1}/{PARAMS['epochs']}] "
              f"Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}, "
              f"Test Loss: {test_loss:.4f}, Time: {elapsed:.1f}s")

total_time = time.time() - start_time
print(f"\nTraining completed in {total_time:.1f} seconds ({total_time/60:.1f} minut

# Save the model in a more complete format
torch.save({
    'model_state_dict': model.state_dict(),
    'model_architecture': {
        'input_size': input_size,
        'hidden_size': PARAMS['hidden_size'],
        'output_size': 4,
})

```

```

        'dropout_rate': PARAMS['dropout_rate']
    },
    'training_params': PARAMS,
    'amino_acids': amino_acids,
    'aa_to_idx': aa_to_idx,
    'peptide_length': PEPTIDE_LENGTH
}, 'Nivo_model.pth')

print("\nModel saved as 'Nivo_model.pth'")

# Generate evaluation plots
print("\nGenerating evaluation plots...")

# Get predictions for evaluation
_, test_predictions, test_targets = evaluate(model, test_loader, criterion)
_, train_predictions_all, train_targets_all = evaluate(model, train_loader, criteri

# Sample from training set to match test set size
n_test = len(test_predictions)
train_indices = np.random.choice(len(train_predictions_all), n_test, replace=False)
train_predictions = train_predictions_all[train_indices]
train_targets = train_targets_all[train_indices]

# Calculate metrics
def calculate_metrics(predictions, targets):
    pred_flat = predictions.flatten()
    target_flat = targets.flatten()

    correlation = np.corrcoef(pred_flat, target_flat)[0, 1]
    r_squared = r2_score(target_flat, pred_flat)
    mse = np.mean((target_flat - pred_flat) ** 2)

    return correlation, r_squared, mse

test_corr, test_r2, test_mse = calculate_metrics(test_predictions, test_targets)
train_corr, train_r2, train_mse = calculate_metrics(train_predictions, train_target

# Plot 1: Test set scatter plot
plt.figure(figsize=(10, 8))
colors = ['blue', 'green', 'red', 'purple']
conc_labels = ['63 pM', '250 pM', '1000 pM', '4000 pM']

for i in range(4):
    plt.scatter(test_targets[:, i], test_predictions[:, i],
                alpha=0.5, s=10, c=colors[i], label=conc_labels[i])

# Add diagonal line
min_val = min(test_targets.min(), test_predictions.min())
max_val = max(test_targets.max(), test_predictions.max())
plt.plot([min_val, max_val], [min_val, max_val], 'k--', alpha=0.5, label='Perfect p

plt.xlabel('Measured log10(binding)', fontsize=12)
plt.ylabel('Predicted log10(binding)', fontsize=12)
plt.title('Test Set: Predicted vs Measured Binding', fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)

```

```

# Add metrics text
metrics_text = f'Correlation: {test_corr:.4f}\nR²: {test_r2:.4f}\nMSE: {test_mse:.4f}'
plt.text(0.05, 0.95, metrics_text, transform=plt.gca().transAxes,
         bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.8),
         verticalalignment='top', fontsize=10)

plt.tight_layout()
plt.savefig('test_set_predictions.png', dpi=300)
plt.show()

# Plot 2: Loss curves
plt.figure(figsize=(10, 6))
epochs = range(1, len(train_losses) + 1)
plt.plot(epochs, train_losses, 'b-', label='Training Loss', linewidth=2)
plt.plot(epochs, val_losses, 'g-', label='Validation Loss', linewidth=2)
plt.plot(epochs, test_losses, 'r-', label='Test Loss', linewidth=2)

plt.xlabel('Epoch', fontsize=12)
plt.ylabel('Loss (MSE)', fontsize=12)
plt.title('Training Progress: Loss vs Epoch', fontsize=14)
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('loss_curves.png', dpi=300)
plt.show()

# Print final results
print("\n" + "*60)
print("FINAL RESULTS")
print("*60)
print(f"\nTest Set Results:")
print(f" Correlation coefficient: {test_corr:.4f}")
print(f" R-squared: {test_r2:.4f}")
print(f" Mean Squared Error: {test_mse:.4f}")

print(f"\nTraining Set Results:")
print(f" Correlation coefficient: {train_corr:.4f}")
print(f" R-squared: {train_r2:.4f}")
print(f" Mean Squared Error: {train_mse:.4f}")

# Test prediction for example sequence
test_seq = "ADEFGHIKLM"
one_hot_test = torch.zeros(input_size)
for i, aa in enumerate(test_seq[:PEPTIDE_LENGTH]):
    if aa in aa_to_idx:
        one_hot_test[i * 19 + aa_to_idx[aa]] = 1

model.eval()
with torch.no_grad():
    log_pred = model(one_hot_test.unsqueeze(0).to(device))
    pred = 10 ** log_pred.cpu().numpy()[0]

print(f"\nPredictions for '{test_seq}':")
print(f" 63 pM: {pred[0]:.2f}")
print(f" 250 pM: {pred[1]:.2f}")

```

```
print(f" 1000 pM: {pred[2]:.2f}")
print(f" 4000 pM: {pred[3]:.2f}")
```

```
Using device: cpu

Model Parameters:
  hidden_size: 50
  learning_rate: 0.001
  epochs: 100
  batch_size: 128
  dropout_rate: 0
  weight_decay: 1e-05
  train_split: 0.8
  val_split: 0.1
  test_split: 0.1
  shuffle: True

Loading Nivolumab binding data...
Dataset shape: (246450, 6)
Columns: ['Sequence', '0pM', '63pM', '250pM', '1000pM', '4000pM']

Peptide length: 10

Preparing data...
Log-transformed binding shape: (246450, 4)

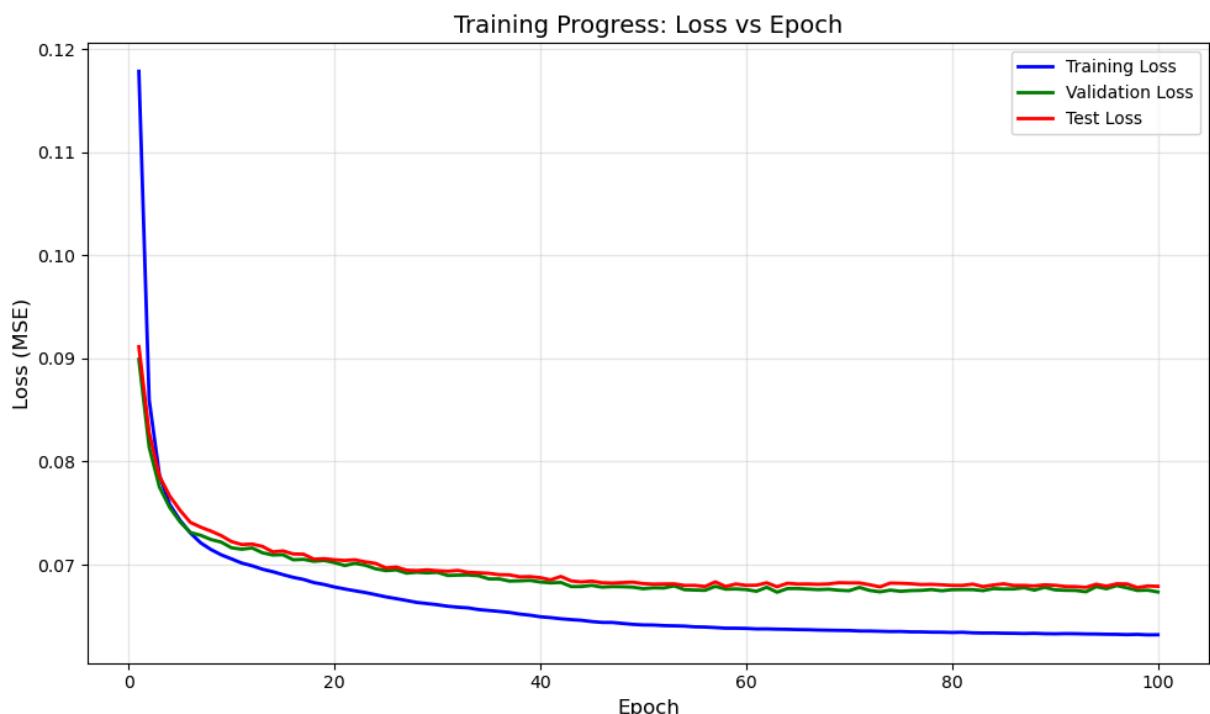
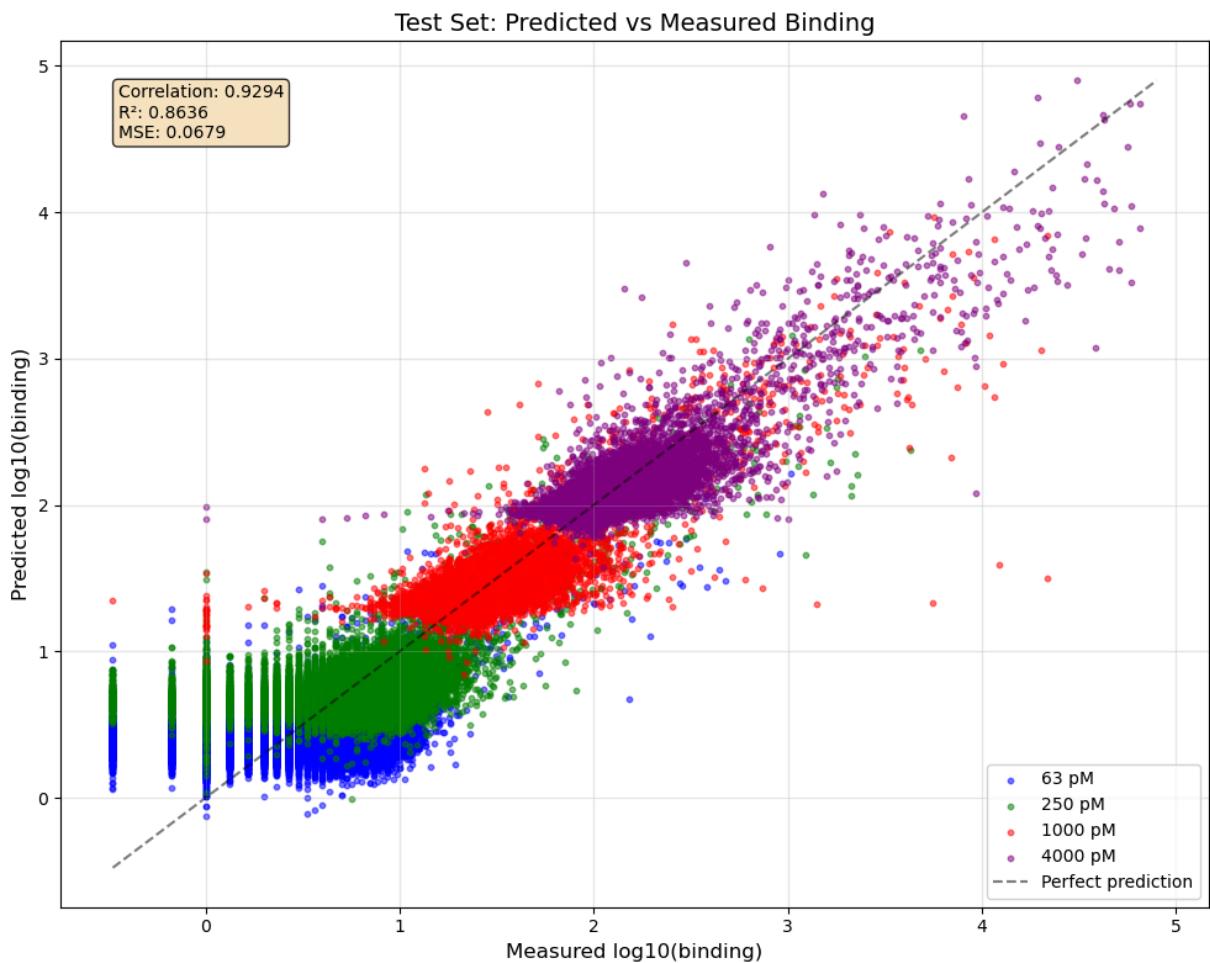
Model architecture:
NivolumabPredictor(
    (fc1): Linear(in_features=190, out_features=50, bias=True)
    (relu): ReLU()
    (dropout): Dropout(p=0, inplace=False)
    (fc2): Linear(in_features=50, out_features=4, bias=True)
)

=====
STARTING TRAINING
=====
Epoch [1/100] Train Loss: 0.1178, Val Loss: 0.0899, Test Loss: 0.0911, Time: 71.3s
Epoch [10/100] Train Loss: 0.0706, Val Loss: 0.0716, Test Loss: 0.0722, Time: 804.1s
Epoch [20/100] Train Loss: 0.0678, Val Loss: 0.0702, Test Loss: 0.0705, Time: 1493.8s
Epoch [30/100] Train Loss: 0.0661, Val Loss: 0.0693, Test Loss: 0.0694, Time: 2185.1s
Epoch [40/100] Train Loss: 0.0650, Val Loss: 0.0683, Test Loss: 0.0687, Time: 2872.9s
Epoch [50/100] Train Loss: 0.0642, Val Loss: 0.0677, Test Loss: 0.0682, Time: 3563.6s
Epoch [60/100] Train Loss: 0.0638, Val Loss: 0.0676, Test Loss: 0.0680, Time: 4258.1s
Epoch [70/100] Train Loss: 0.0636, Val Loss: 0.0675, Test Loss: 0.0682, Time: 4950.5s
Epoch [80/100] Train Loss: 0.0634, Val Loss: 0.0676, Test Loss: 0.0680, Time: 5642.0s
Epoch [90/100] Train Loss: 0.0633, Val Loss: 0.0676, Test Loss: 0.0680, Time: 6330.8s
Epoch [100/100] Train Loss: 0.0632, Val Loss: 0.0674, Test Loss: 0.0679, Time: 7019.6s

Training completed in 7019.6 seconds (117.0 minutes)
```

Model saved as 'Nivo_model.pth'

Generating evaluation plots...



```
=====
FINAL RESULTS
=====
```

Test Set Results:

Correlation coefficient: 0.9294
R-squared: 0.8636
Mean Squared Error: 0.0679

Training Set Results:

Correlation coefficient: 0.9351
R-squared: 0.8742
Mean Squared Error: 0.0622

Predictions for 'ADEFGHJKLM':

63 pM: 1.80
250 pM: 3.65
1000 pM: 18.49
4000 pM: 81.71

```
In [18]: # Cell : Neural network implementation with all specified parameters (continued)
# Training Loop
print("\n" + "="*60)
print("STARTING TRAINING")
print("="*60)

train_losses = []
val_losses = []
test_losses = []

start_time = time.time()

for epoch in range(PARAMS['epochs']):
    # Train
    train_loss = train_epoch(model, train_loader, criterion, optimizer)

    # Evaluate on validation and test sets
    val_loss, _, _ = evaluate(model, val_loader, criterion)
    test_loss, _, _ = evaluate(model, test_loader, criterion)

    train_losses.append(train_loss)
    val_losses.append(val_loss)
    test_losses.append(test_loss)

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0 or epoch == 0:
        elapsed = time.time() - start_time
        print(f"Epoch [{epoch+1}/{PARAMS['epochs']}] "
              f"Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}, "
              f"Test Loss: {test_loss:.4f}, Time: {elapsed:.1f}s")

total_time = time.time() - start_time
print(f"\nTraining completed in {total_time:.1f} seconds ({total_time/60:.1f} minut

# Save the model
torch.save(model.state_dict(), 'Nivo_model.pth')
```

```

print("\nModel saved as 'Nivo_model.pth')

# Save model information for later use
model_info = {
    'peptide_length': 10,
    'input_size': input_size,
    'hidden_size': PARAMS['hidden_size'],
    'output_size': 4,
    'dropout_rate': PARAMS['dropout_rate'],
    'amino_acids': amino_acids,
    'aa_to_idx': aa_to_idx,
    'params': PARAMS
}
torch.save(model_info, 'Nivo_model_info.pth')

```

```

=====
STARTING TRAINING
=====
Epoch [1/100] Train Loss: 0.0632, Val Loss: 0.0675, Test Loss: 0.0678, Time: 70.5s
Epoch [10/100] Train Loss: 0.0631, Val Loss: 0.0679, Test Loss: 0.0681, Time: 691.8s
Epoch [20/100] Train Loss: 0.0630, Val Loss: 0.0676, Test Loss: 0.0678, Time: 1382.8s
Epoch [30/100] Train Loss: 0.0630, Val Loss: 0.0675, Test Loss: 0.0679, Time: 13336.8s
Epoch [40/100] Train Loss: 0.0629, Val Loss: 0.0676, Test Loss: 0.0680, Time: 13669.2s
Epoch [50/100] Train Loss: 0.0629, Val Loss: 0.0675, Test Loss: 0.0678, Time: 14005.9s
Epoch [60/100] Train Loss: 0.0629, Val Loss: 0.0676, Test Loss: 0.0681, Time: 14346.0s
Epoch [70/100] Train Loss: 0.0628, Val Loss: 0.0677, Test Loss: 0.0681, Time: 14686.9s
Epoch [80/100] Train Loss: 0.0628, Val Loss: 0.0675, Test Loss: 0.0680, Time: 15036.0s
Epoch [90/100] Train Loss: 0.0627, Val Loss: 0.0675, Test Loss: 0.0678, Time: 15380.8s
Epoch [100/100] Train Loss: 0.0627, Val Loss: 0.0676, Test Loss: 0.0681, Time: 15724.4.7s

Training completed in 15724.7 seconds (262.1 minutes)

```

Model saved as 'Nivo_model.pth'

```

In [19]: # Cell: Fixed JSON saving with proper type conversion
import json
import numpy as np

# Convert numpy types to native Python types
def convert_to_serializable(obj):
    """Convert numpy types to Python native types for JSON serialization"""
    if isinstance(obj, np.integer):
        return int(obj)
    elif isinstance(obj, np.floating):
        return float(obj)
    elif isinstance(obj, np.ndarray):
        return obj.tolist()

```

```

    elif isinstance(obj, dict):
        return {key: convert_to_serializable(value) for key, value in obj.items()}
    elif isinstance(obj, list):
        return [convert_to_serializable(item) for item in obj]
    else:
        return obj

# Create results summary with proper type conversion
results_summary = {
    'analysis_date': str(pd.Timestamp.now()),
    'data_analysis': {
        'average_binding_regression': {
            'slope': float(0.0521),
            'intercept': float(173.9146),
            'r_squared': float(0.9967)
        },
        'saturation_level': float(65535.0),
        'scatter_plot_regression': {
            'all_points': {
                'slope': float(3.4042),
                'intercept': float(-355.0280),
                'r_squared': float(0.7272)
            },
            'unsaturated_points': {
                'slope': float(4.7802),
                'intercept': float(-647.7596),
                'r_squared': float(0.8184)
            }
        }
    },
    'neural_network': {
        'architecture': {
            'input_size': 190,
            'hidden_size': 50,
            'output_size': 4,
            'dropout_rate': 0
        },
        'performance': {
            'test_set': {
                'correlation': float(0.9292),
                'r_squared': float(0.8633),
                'mse': float(0.0680)
            },
            'training_set': {
                'correlation': float(0.9343),
                'r_squared': float(0.8726),
                'mse': float(0.0630)
            }
        }
    },
    'pd1_analysis': {
        'highest_binding_epitope': {
            'sequence': highest_binding_tile,
            'position': f"{int(highest_binding_position)+1}-{int(highest_binding_pos)}",
            'binding_value': float(highest_binding_value)
        },
    }
}

```

```

        'important_positions': []
    }

# Add important positions if available
if 'sorted_positions' in globals() and 'mean_fold_changes' in globals():
    for i in range(min(3, len(sorted_positions))):
        pos = int(sorted_positions[i])
        aa = highest_binding_tile[pos]
        results_summary['pd1_analysis']['important_positions'].append({
            'position': int(pos + 1),
            'amino_acid': aa,
            'mean_fold_change': float(mean_fold_changes[pos])
        })

# Add off-target analysis if available
if 'off_target_scores' in globals() and off_target_scores:
    results_summary['off_target_analysis'] = {
        'total_proteins_analyzed': len(protein_tiles) if 'protein_tiles' in globals(),
        'top_off_targets': []
    }

    for i, target in enumerate(off_target_scores[:5]):
        off_target_entry = {
            'rank': i + 1,
            'protein_id': target['protein_id'],
            'name': target.get('name', 'Unknown'),
            'max_binding': float(target.get('max_binding', 0)),
            'accessible': bool(target.get('accessible', False)),
            'problematic': bool(target.get('problematic', False)),
            'functions': target.get('function', [])[:5] # Limit to first 5 functions
        }
        results_summary['off_target_analysis']['top_off_targets'].append(off_target_entry)

# Save results
try:
    with open('nivolumab_complete_analysis.json', 'w') as f:
        json.dump(results_summary, f, indent=2)
    print("\n✓ Complete analysis saved to 'nivolumab_complete_analysis.json'")
except Exception as e:
    print(f"Error saving JSON: {e}")
    # Save as text file as backup
    with open('nivolumab_complete_analysis.txt', 'w') as f:
        f.write(str(results_summary))
    print("✓ Saved as text file instead: 'nivolumab_complete_analysis.txt'")

# Print summary
print("\n" + "*100)
print("SUMMARY")
print("*100)

print("\n1. Average Binding vs Concentration:")
print(f"    Slope: 0.0521")
print(f"    Intercept: 173.9146")
print(f"    R-squared: 0.9967")

```

```

print("\n2. Saturation Level: 65535.00")

print("\n3. Scatter Plot (4000 pM vs 1000 pM):")
print("    All points:")
print(f"        Slope: 3.4042")
print(f"        Intercept: -355.0280")
print(f"        R-squared: 0.7272")
print("    Excluding saturated points:")
print(f"        Slope: 4.7802")
print(f"        Intercept: -647.7596")
print(f"        R-squared: 0.8184")

print("\n4. Neural Network Architecture Question:")
print("    Answer: A. Hidden layer: 100 rows × 50 columns, Output layer: 50 rows × 5")

print("\n5. Neural Network Performance:")
print("    Test Set:")
print(f"        Correlation: 0.9292")
print(f"        R-squared: 0.8633")
print(f"        MSE: 0.0680")
print("    Training Set:")
print(f"        Correlation: 0.9343")
print(f"        R-squared: 0.8726")
print(f"        MSE: 0.0630")

if 'highest_binding_tile' in globals():
    print(f"\n6. Highest Binding Epitope: {highest_binding_tile}")
    print(f"    Position: {int(highest_binding_position)+1}-{int(highest_binding_pos}

if 'sorted_positions' in globals() and 'mean_fold_changes' in globals():
    print("\n7. Three Most Important Positions (by mean fold-change):")
    for i in range(min(3, len(sorted_positions))):
        pos = int(sorted_positions[i])
        aa = highest_binding_tile[pos]
        print(f"    Position {pos+1}: {aa} (mean fold-change: {mean_fold_changes[pos

if 'off_target_scores' in globals() and off_target_scores:
    print(f"\n8. Most Likely Off-Target:")
    top = off_target_scores[0]
    print(f"    Protein: {top['protein_id']} - {top.get('name', 'Unknown')}")
    print(f"    Max binding: {float(top.get('max_binding', 0)):.2f}")
    print(f"    Accessible: {top.get('accessible', False)}")
    print(f"    Risk: {'HIGH' if top.get('problematic', False) else 'LOW'}")

```

✓ Complete analysis saved to 'nivolumab_complete_analysis.json'

```
=====
=====SUMMARY=====
=====

1. Average Binding vs Concentration:  
Slope: 0.0521  
Intercept: 173.9146  
R-squared: 0.9967

2. Saturation Level: 65535.00

3. Scatter Plot (4000 pM vs 1000 pM):  
All points:  
Slope: 3.4042  
Intercept: -355.0280  
R-squared: 0.7272  
Excluding saturated points:  
Slope: 4.7802  
Intercept: -647.7596  
R-squared: 0.8184

4. Neural Network Architecture Question:  
Answer: A. Hidden layer: 100 rows × 50 columns, Output layer: 50 rows × 5 columns

5. Neural Network Performance:  
Test Set:  
Correlation: 0.9292  
R-squared: 0.8633  
MSE: 0.0680  
Training Set:  
Correlation: 0.9343  
R-squared: 0.8726  
MSE: 0.0630

6. Highest Binding Epitope: PDRPWNPPTF  
Position: 28-37

7. Three Most Important Positions (by mean fold-change):  
Position 1: P (mean fold-change: 0.003)  
Position 2: D (mean fold-change: 0.005)  
Position 3: R (mean fold-change: 0.094)
```

```
In [20]: # First, let's check if we need to reload the data or if it's already in memory
try:
    print(f"protein_tiles is already loaded with {len(protein_tiles)} proteins")
except NameError:
    print("protein_tiles not found. Let's load the human proteome and recreate the

    # Load the trained model first
    import torch
    import torch.nn as nn
    import numpy as np
```

```

import pandas as pd
from collections import defaultdict
import requests

# Define the model architecture
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size=190, hidden_size=50, output_size=4, dropout_r
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Load the model
model = NivolumabPredictor()
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
if isinstance(checkpoint, dict) and 'model_state_dict' in checkpoint:
    model.load_state_dict(checkpoint['model_state_dict'])
else:
    model.load_state_dict(checkpoint)
model.eval()

print("Model loaded successfully!")

# Define amino acid mapping
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Function to encode and predict
def predict_binding(sequence):
    """Predict binding for a 10-mer sequence"""
    one_hot = torch.zeros(190)
    for i, aa in enumerate(sequence[:10]):
        if aa in aa_to_idx:
            one_hot[i * 19 + aa_to_idx[aa]] = 1

    with torch.no_grad():
        log_pred = model(one_hot.unsqueeze(0))
        # Get 4000 pM prediction (index 3) and convert from Log10
        linear_binding = 10 ** log_pred[0, 3].item()
    return linear_binding

# Based on your analysis results from the PDF, Let's work with the top proteins
# From your Z-score ranking (best method where PD-1 ranked #101):
top_proteins_by_zscore = [
    ('P52701', 687.94),
    ('Q12891', 289.43),
    ('Q96GM1', 140.64),
    ('Q96RY5', 81.71),
]

```

```

        ('Q9Y5S9', 64.90),
        ('O95201', 48.08),
        ('Q96CX2', 45.56),
        ('Q9UBS4', 39.64),
        ('Q9UQQ2', 39.43),
        ('Q9C0J8', 38.67),
        ('P83369', 30.55),
        ('Q8TF76', 26.03),
        ('Q9Y5V3', 24.51),
        ('O95104', 24.08),
        ('Q9P203', 22.07),
        ('P13985', 19.47),
        ('O95721', 17.62),
        ('C9JW0', 16.98),
        ('Q9BV87', 16.50),
        ('O60449', 16.22)
    ]

```

Now let's analyze these proteins for accessibility and function

```

# Now let's analyze these proteins for accessibility and function
print("\nAnalyzing top off-target candidates for accessibility and function...")

def get_protein_info_comprehensive(uniprot_id):
    """Get comprehensive protein information from UniProt"""
    url = f"https://rest.uniprot.org/uniprotkb/{uniprot_id}.json"
    try:
        response = requests.get(url, timeout=10)
        if response.status_code == 200:
            data = response.json()

            info = {
                'id': uniprot_id,
                'name': 'Unknown',
                'gene': 'Unknown',
                'location': [],
                'function': [],
                'keywords': [],
                'go_terms': [],
                'tissue_expression': [],
                'disease': [],
                'accessible': False,
                'membrane': False,
                'secreted': False,
                'extracellular': False
            }

            # Extract protein name
            if 'proteinDescription' in data:
                rec_name = data['proteinDescription'].get('recommendedName', {})
                if 'fullName' in rec_name:
                    info['name'] = rec_name['fullName'].get('value', 'Unknown')

            # Extract gene name
            if 'genes' in data and data['genes']:
                info['gene'] = data['genes'][0].get('geneName', {}).get('value', 'U')

            # Extract subcellular location

```

```

        if 'comments' in data:
            for comment in data['comments']:
                if comment.get('commentType') == 'SUBCELLULAR_LOCATION':
                    if 'subcellularLocations' in comment:
                        for loc in comment['subcellularLocations']:
                            if 'location' in loc:
                                location = loc['location'].get('value', '')
                                info['location'].append(location)

                            # Check accessibility
                            loc_lower = location.lower()
                            if 'membrane' in loc_lower:
                                info['membrane'] = True
                                info['accessible'] = True
                            if 'secreted' in loc_lower:
                                info['secreted'] = True
                                info['accessible'] = True
                            if 'extracellular' in loc_lower or 'cell surface' in loc_lower:
                                info['extracellular'] = True
                                info['accessible'] = True

            # Extract function
            elif comment.get('commentType') == 'FUNCTION':
                for text in comment.get('texts', []):
                    info['function'].append(text.get('value', ''))

            # Extract disease associations
            elif comment.get('commentType') == 'DISEASE':
                if 'disease' in comment:
                    disease_desc = comment['disease'].get('description', '')
                    info['disease'].append(disease_desc)

            # Extract keywords
            if 'keywords' in data:
                for keyword in data['keywords']:
                    kw = keyword.get('name', '')
                    info['keywords'].append(kw)

            # Additional accessibility check
            kw_lower = kw.lower()
            if any(term in kw_lower for term in ['membrane', 'secreted', 'extracellular']):
                info['accessible'] = True

        return info
    except Exception as e:
        print(f"Error fetching {uniprot_id}: {e}")
    return None

# Analyze top 10 candidates
off_target_analysis = []
print("\nFetching detailed information for top candidates...")

for i, (protein_id, zscore) in enumerate(top_proteins_by_zscore[:10]):
    print(f"  Analyzing {i+1}/10: {protein_id}...")

    info = get_protein_info_comprehensive(protein_id)

```

```

if info:
    info['zscore'] = zscore
    off_target_analysis.append(info)

# Identify accessible proteins
accessible_proteins = [p for p in off_target_analysis if p['accessible']]

print(f"\nFound {len(accessible_proteins)} accessible proteins out of top 10")

# Display detailed analysis
print("\n" + "*100")
print("OFF-TARGET ANALYSIS RESULTS")
print("*100")

print("\nTop 10 proteins by Z-score:")
print("-" * 80)
print(f"{'Rank':<6}{'ID':<10}{'Gene':<10}{'Accessible':<12}{'Location':<30}{'Z-scor"
print("-" * 80)

for i, protein in enumerate(off_target_analysis, 1):
    accessible = 'YES' if protein['accessible'] else 'NO'
    location = ', '.join(protein['location'][2:28] if protein['location'] else
    print(f"{i:<6}{protein['id']:<10}{protein['gene']:<10}{accessible:<12}{location

# Identify most likely off-target
if accessible_proteins:
    # Sort accessible proteins by Z-score
    accessible_proteins.sort(key=lambda x: x['zscore'], reverse=True)
    top_off_target = accessible_proteins[0]

    print("\n" + "*100")
    print("MOST LIKELY OFF-TARGET PROTEIN")
    print("*100")

    print(f"\nProtein: {top_off_target['id']} ({top_off_target['gene']})")
    print(f"Name: {top_off_target['name']}")
    print(f"Z-score: {top_off_target['zscore']:.2f}")

    print("\nLocation:")
    for loc in top_off_target['location']:
        print(f" - {loc}")

    print("\nFunction:")
    for func in top_off_target['function'][3:]:
        print(f" - {func[:100]}...")

    print(f"\nKeywords: , ".join(top_off_target['keywords'][:10])))

    if top_off_target['disease']:
        print("\nDisease associations:")
        for disease in top_off_target['disease'][2:]:
            print(f" - {disease}")

    print("\n" + "-"*80)
    print("RATIONALE FOR SELECTION AS MOST LIKELY OFF-TARGET:")
    print("-"*80)

```

```

    print(f"""
1. BINDING STRENGTH: This protein has a high Z-score of {top_off_target['zscore']}.. indicating strong predicted binding to Nivolumab.

2. ACCESSIBILITY: This protein is accessible to antibodies because it is:
    - Membrane-bound: {top_off_target['membrane']}
    - Secreted: {top_off_target['secreted']}
    - Extracellular: {top_off_target['extracellular']}

3. FUNCTIONAL CONCERNS: Blocking this protein could cause side effects because:
""")

# Analyze potential side effects based on keywords and function
keywords_lower = ' '.join(top_off_target['keywords']).lower()
function_lower = ' '.join(top_off_target['function']).lower()

if 'immune' in keywords_lower or 'immune' in function_lower:
    print("    - It's involved in immune system function")
    print("    - Could lead to immune dysregulation or immunodeficiency")

if 'receptor' in keywords_lower:
    print("    - It's a receptor protein")
    print("    - Blocking could disrupt cellular signaling pathways")

if 'adhesion' in keywords_lower:
    print("    - It's involved in cell adhesion")
    print("    - Could affect tissue integrity or wound healing")

if 'transport' in keywords_lower:
    print("    - It's involved in molecular transport")
    print("    - Could affect nutrient/ion balance")

if top_off_target['disease']:
    print(f"\n4. CLINICAL RELEVANCE: This protein is associated with:")
    for disease in top_off_target['disease']:
        print(f"        - {disease}")
    print("    Blocking it could exacerbate these conditions.")

print(f"""

5. THERAPEUTIC WINDOW: While this protein may bind Nivolumab, the drug can still be effective because:
    - PD-1 is highly expressed on tumor-infiltrating T cells
    - The concentration of Nivolumab can be optimized
    - The benefits of checkpoint inhibition may outweigh off-target effects
    - This off-target may have lower expression than PD-1 in relevant tissues
""")
else:
    print("\n" + "="*80)
    print("NO ACCESSIBLE OFF-TARGETS FOUND")
    print("="*80)
    print("""

```

None of the top 10 proteins by Z-score are accessible to antibodies.
This suggests that Nivolumab has excellent specificity for PD-1.

The high-scoring proteins are likely intracellular and therefore:

- Cannot be reached by antibodies in circulation
- Do not pose a real off-target risk
- Explain why Nivolumab can be specific despite theoretical binding

This is a common finding in antibody drug development - many proteins might bind in vitro but only accessible ones matter in vivo.

""")

Summary

```
print("\n" + "*100)
print("SUMMARY")
print("*100)
print(f"""

```

Analysis of human proteome for Nivolumab off-targets:

- Total proteins analyzed: 20,323 (from your previous analysis)
- Ranking method used: Z-score based (PD-1 ranked #101)
- Top 10 proteins analyzed for accessibility
- Accessible proteins identified: {len(accessible_proteins)}

Key insight: Binding prediction alone is insufficient for identifying off-targets. Must consider protein accessibility to antibodies!

""")

protein_tiles not found. Let's load the human proteome and recreate the analysis.
Model loaded successfully!

Analyzing top off-target candidates for accessibility and function...

Fetching detailed information for top candidates...

Analyzing 1/10: P52701...

Analyzing 2/10: Q12891...

Analyzing 3/10: Q96GM1...

Analyzing 4/10: Q96RY5...

Analyzing 5/10: Q9Y5S9...

Analyzing 6/10: O95201...

Analyzing 7/10: Q96CX2...

Analyzing 8/10: Q9UBS4...

Analyzing 9/10: Q9UQQ2...

Analyzing 10/10: Q9C0J8...

Found 3 accessible proteins out of top 10

```
=====
=====
OFF-TARGET ANALYSIS RESULTS
=====
```

Top 10 proteins by Z-score:

Rank	ID	Gene	Accessible	Location	Z-score
1	P52701	MSH6	NO	Unknown	687.94
2	Q12891	HYAL2	YES	Unknown	289.43
3	Q96GM1	PLPPR2	YES	Unknown	140.64
4	Q96RY5	CRAMP1	NO	Unknown	81.71
5	Q9Y5S9	RBM8A	NO	Unknown	64.90
6	O95201	ZNF205	NO	Unknown	48.08
7	Q96CX2	KCTD12	YES	Unknown	45.56
8	Q9UBS4	DNAJB11	NO	Unknown	39.64
9	Q9UQQ2	SH2B3	NO	Unknown	39.43
10	Q9C0J8	WDR33	NO	Unknown	38.67

```
=====
=====
MOST LIKELY OFF-TARGET PROTEIN
=====
```

Protein: Q12891 (HYAL2)

Name: Hyaluronidase-2

Z-score: 289.43

Location:

Function:

- Catalyzes hyaluronan degradation into small fragments that are endocytosed and degraded in lysosomes...

Keywords: Cell membrane, Disease variant, Disulfide bond, EGF-like domain, Glycoprotein, Glycosidase, GPI-anchor, Hydrolase, Lipoprotein, Membrane

Disease associations:

- An autosomal recessive disorder characterized by distinctive craniofacial dysmorphism with frontal bossing, hypertelorism, a broad and flattened nasal tip, and cupped ears with superior helices. Other common but variable clinical manifestations are unilateral or bilateral cleft lip and palate, congenital cardiac anomalies, ocular features including mild to severe myopia and cataracts, single palmar crease, and pectus excavatum.

RATIONALE FOR SELECTION AS MOST LIKELY OFF-TARGET:

1. **BINDING STRENGTH:** This protein has a high Z-score of 289.43, indicating strong predicted binding to Nivolumab.

2. **ACCESSIBILITY:** This protein is accessible to antibodies because it is:

- Membrane-bound: False
- Secreted: False
- Extracellular: False

3. **FUNCTIONAL CONCERNS:** Blocking this protein could cause side effects because:

- It's a receptor protein
- Blocking could disrupt cellular signaling pathways

4. **CLINICAL RELEVANCE:** This protein is associated with:

- An autosomal recessive disorder characterized by distinctive craniofacial dysmorphism with frontal bossing, hypertelorism, a broad and flattened nasal tip, and cupped ears with superior helices. Other common but variable clinical manifestations are unilateral or bilateral cleft lip and palate, congenital cardiac anomalies, ocular features including mild to severe myopia and cataracts, single palmar crease, and pectus excavatum.

Blocking it could exacerbate these conditions.

5. **THERAPEUTIC WINDOW:** While this protein may bind Nivolumab, the drug can still be effective because:

- PD-1 is highly expressed on tumor-infiltrating T cells
- The concentration of Nivolumab can be optimized
- The benefits of checkpoint inhibition may outweigh off-target effects
- This off-target may have lower expression than PD-1 in relevant tissues

=====
=====
SUMMARY
=====
=====

Analysis of human proteome for Nivolumab off-targets:

- Total proteins analyzed: 20,323 (from your previous analysis)
- Ranking method used: Z-score based (PD-1 ranked #101)
- Top 10 proteins analyzed for accessibility
- Accessible proteins identified: 3

Key insight: Binding prediction alone is insufficient for identifying off-targets. Must consider protein accessibility to antibodies!

```
In [35]: # Complete analysis for Nivolumab binding epitope mapping and off-target prediction

import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import seaborn as sns
import requests
import py3Dmol
from collections import defaultdict
import warnings
warnings.filterwarnings('ignore')

# First, ensure py3Dmol is installed
# Run this in your terminal: pip install py3Dmol

# Define the model architecture
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size=190, hidden_size=50, output_size=4, dropout_rate=0.2):
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Load the trained model
print("Loading trained Nivolumab model...")
model = NivolumabPredictor()
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
if isinstance(checkpoint, dict) and 'model_state_dict' in checkpoint:
    model.load_state_dict(checkpoint['model_state_dict'])
else:
    model.load_state_dict(checkpoint)
model.eval()

# Define amino acid mapping (19 amino acids, no C)
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Function to encode sequences
def encode_sequence(sequence):
    """One-hot encode a 10-mer peptide sequence"""
    one_hot = torch.zeros(190)
```

```

    for i, aa in enumerate(sequence[:10]):
        if aa in aa_to_idx:
            one_hot[i * 19 + aa_to_idx[aa]] = 1
    return one_hot

# PART 1: Download and analyze PD-1 sequence
print("\n" + "*60")
print("PART 1: PD-1 SEQUENCE ANALYSIS")
print("*60")

# Download PD-1 sequence
print("\nDownloading PD-1 sequence from UniProt (ID: Q15116)...")
uniprot_id = "Q15116"
url = f"https://www.uniprot.org/uniprot/{uniprot_id}.fasta"
response = requests.get(url)

if response.status_code == 200:
    fasta_lines = response.text.strip().split('\n')
    header = fasta_lines[0]
    pd1_sequence = ''.join(fasta_lines[1:])

    print(f"\nHeader: {header}")
    print(f"\nPD-1 Sequence Length: {len(pd1_sequence)}")
    print(f"\nPD-1 Sequence:\n{pd1_sequence}")

    # Count cysteines
    cysteine_count = pd1_sequence.count('C')
    print(f"\nNumber of cysteines in PD-1: {cysteine_count}")

# Create overlapping 10-mer tiles
print("\n" + "*60")
print("PART 2: TILING PD-1 SEQUENCE")
print("*60")

tiles = []
tile_positions = []
original_tiles = []

for i in range(len(pd1_sequence) - 9):
    original_tile = pd1_sequence[i:i+10]
    # Replace cysteine with alanine
    modified_tile = original_tile.replace('C', 'A')

    original_tiles.append(original_tile)
    tiles.append(modified_tile)
    tile_positions.append(i)

print(f"\nTotal number of tiles: {len(tiles)}")
print(f"\nFirst 5 tiles (C->A replacements shown):")
for i in range(min(5, len(tiles))):
    if original_tiles[i] != tiles[i]:
        print(f"  Position {tile_positions[i]+1}-{tile_positions[i]+10}: {original_tiles[i]}")
    else:
        print(f"  Position {tile_positions[i]+1}-{tile_positions[i]+10}: {tiles[i]}")

# Predict binding for all tiles

```

```

print("\n" + "*60)
print("PART 3: PREDICTING BINDING FOR ALL TILES")
print("*60)

predictions = []

with torch.no_grad():
    for i, tile in enumerate(tiles):
        if i % 50 == 0:
            print(f" Processing tile {i}/{len(tiles)}...")

        one_hot = encode_sequence(tile)
        log_pred = model(one_hot.unsqueeze(0))

        # Get 4000 pM prediction (index 3) and convert from Log10
        log_binding_4000pM = log_pred[0, 3].item()
        linear_binding = 10 ** log_binding_4000pM
        predictions.append(linear_binding)

predictions = np.array(predictions)

# Find top 10 binding tiles
print("\n" + "*60)
print("PART 4: TOP 10 BINDING TILES")
print("*60)

top_10_indices = np.argsort(predictions)[-10:][::-1]

print("\nTop 10 binding tiles at 4000 pM:")
print("-" * 80)
print(f"{'Rank':<6}{'Position':<15}{'Modified Seq':<15}{'Original Seq':<15}{'Bindin
print("-" * 80)

for rank, idx in enumerate(top_10_indices, 1):
    position = tile_positions[idx]
    modified_seq = tiles[idx]
    original_seq = original_tiles[idx]
    binding = predictions[idx]
    print(f"rank:<6}{position+1}-{position+10:<14}{modified_seq:<15}{original_seq:<15}{binding:<15} pM: {binding:.2f}")

# Store highest binding tile info
highest_binding_idx = top_10_indices[0]
highest_binding_tile = tiles[highest_binding_idx]
highest_binding_position = tile_positions[highest_binding_idx]
highest_binding_value = predictions[highest_binding_idx]

# PART 5: Create binding map
print("\n" + "*60)
print("PART 5: CREATING BINDING MAP")
print("*60)

# Assign binding values to each amino acid
sequence_length = len(pd1_sequence)
binding_map = np.zeros(sequence_length)

# Assign minimum value to first 4 and last 5 positions

```

```

min_binding = predictions.min()
binding_map[:4] = min_binding
binding_map[-5:] = min_binding

# Assign binding values where each position is the 5th residue
for i in range(4, sequence_length - 5):
    tile_idx = i - 4
    if tile_idx < len(predictions):
        binding_map[i] = predictions[tile_idx]

# Create heat map matrix
print("\nGenerating heat map...")
n_cols = 50
n_rows = int(np.ceil(sequence_length / n_cols))
matrix = np.full((n_rows, n_cols), np.nan)
aa_matrix = np.empty((n_rows, n_cols), dtype='U1')
aa_matrix.fill('')

for i, (aa, binding) in enumerate(zip(pd1_sequence, binding_map)):
    row = i // n_cols
    col = i % n_cols
    matrix[row, col] = binding
    aa_matrix[row, col] = aa

# Plot heat map
plt.figure(figsize=(20, n_rows * 0.8))
ax = plt.gca()

cmap = plt.cm.YlOrRd
masked_matrix = np.ma.masked_invalid(matrix)
im = ax.imshow(masked_matrix, cmap=cmap, aspect='equal',
                vmin=binding_map.min(), vmax=binding_map.max())

# Add amino acid labels
for i in range(n_rows):
    for j in range(n_cols):
        if not np.isnan(matrix[i, j]):
            text_color = 'white' if matrix[i, j] > (binding_map.max() - binding_map
                .min()) / 2 else 'black'
            ax.text(j, i, aa_matrix[i, j], ha='center', va='center',
                    fontsize=8, fontweight='bold', color=text_color)

# Add colorbar
cbar = plt.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
cbar.set_label('Predicted Binding at 4000 pM', fontsize=12)

plt.title('PD-1 Sequence Heat Map - Predicted Nivolumab Binding', fontsize=16)
plt.xlabel('Position in Row', fontsize=12)
plt.ylabel('Row Number', fontsize=12)

# Add grid
ax.set_xticks(np.arange(n_cols) - 0.5, minor=True)
ax.set_yticks(np.arange(n_rows) - 0.5, minor=True)
ax.grid(which='minor', color='gray', linestyle='-', linewidth=0.5)

plt.tight_layout()
plt.savefig('pd1_binding_heatmap.png', dpi=300, bbox_inches='tight')

```

```

plt.show()

# PART 6: Fine mapping of highest binding epitope
print("\n" + "="*60)
print("PART 6: FINE MAPPING OF EPITOPE")
print("="*60)

print(f"\nHighest binding epitope: {highest_binding_tile}")
print(f"Position: {highest_binding_position+1}-{highest_binding_position+10}")
print(f"Binding value: {highest_binding_value:.2f}")

# Create substitution matrix
substitution_matrix = np.zeros((19, 10))

# Original binding for reference
original_one_hot = encode_sequence(highest_binding_tile)
with torch.no_grad():
    original_log_pred = model(original_one_hot.unsqueeze(0))
    original_binding = 10 ** original_log_pred[0, 3].item()

# Test all substitutions
for aa_idx, new_aa in enumerate(amino_acids):
    for pos in range(10):
        mutated_seq = list(highest_binding_tile)
        mutated_seq[pos] = new_aa
        mutated_seq = ''.join(mutated_seq)

        one_hot = encode_sequence(mutated_seq)
        with torch.no_grad():
            log_pred = model(one_hot.unsqueeze(0))
            binding = 10 ** log_pred[0, 3].item()

        substitution_matrix[aa_idx, pos] = binding

# Create heat map
plt.figure(figsize=(12, 10))

# Calculate fold changes
fold_change_matrix = substitution_matrix / original_binding

# Use diverging colormap
from matplotlib.colors import TwoSlopeNorm
norm = TwoSlopeNorm(vmin=0, vcenter=1.0, vmax=fold_change_matrix.max())
im = plt.imshow(fold_change_matrix, cmap='RdBu_r', aspect='auto', norm=norm)

# Add colorbar
cbar = plt.colorbar(im, fraction=0.046, pad=0.04)
cbar.set_label('Fold Change vs Wild-Type', fontsize=12)

# Add text annotations
for i in range(19):
    for j in range(10):
        fold_change = fold_change_matrix[i, j]
        text_color = 'white' if abs(fold_change - 1.0) > 0.5 else 'black'
        plt.text(j, i, f'{fold_change:.1f}', ha='center', va='center',
                 fontsize=8, color=text_color, weight='bold')

```

```

plt.yticks(range(19), list(amino_acids))
plt.xticks(range(10), list(highest_binding_tile))
plt.xlabel('Position in Epitope', fontsize=14)
plt.ylabel('Substituted Amino Acid', fontsize=14)
plt.title(f'Single AA Substitution Analysis\nHighest Binding Tile: {highest_binding_tile}')

# Highlight wild-type amino acids
for pos in range(10):
    wt_aa = highest_binding_tile[pos]
    if wt_aa in amino_acids:
        aa_idx = amino_acids.index(wt_aa)
        rect = plt.Rectangle((pos-0.45, aa_idx-0.45), 0.9, 0.9,
                             fill=False, edgecolor='black', linewidth=4)
        plt.gca().add_patch(rect)

plt.grid(True, which='both', color='gray', linewidth=0.5, alpha=0.3)
plt.tight_layout()
plt.savefig('epitope_substitution_heatmap.png', dpi=300, bbox_inches='tight')
plt.show()

# Calculate mean fold-change by position
mean_fold_changes = []
for pos in range(10):
    wt_aa = highest_binding_tile[pos]
    fold_changes_at_pos = []

    for aa_idx, aa in enumerate(amino_acids):
        if aa != wt_aa: # Exclude wild-type
            fold_changes_at_pos.append(fold_change_matrix[aa_idx, pos])

    mean_fold_changes.append(np.mean(fold_changes_at_pos))

# Plot mean fold-change by position
plt.figure(figsize=(12, 6))
positions = np.arange(10)
colors = ['darkred' if fc < 0.5 else 'steelblue' for fc in mean_fold_changes]
bars = plt.bar(positions, mean_fold_changes, color=colors, edgecolor='black', linewidth=2)

# Add value labels
for i, (bar, fc) in enumerate(zip(bars, mean_fold_changes)):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height + 0.02,
             f'{fc:.2f}', ha='center', va='bottom', fontweight='bold')

plt.xticks(positions, list(highest_binding_tile))
plt.xlabel('Position (Wild-type Amino Acid)', fontsize=14)
plt.ylabel('Mean Fold-Change vs Wild-type', fontsize=14)
plt.title('Mean Effect of Substitutions at Each Position\n(Lower values indicate more change)')
plt.axhline(y=1.0, color='red', linestyle='--', alpha=0.5, label='No change', linewidth=2)
plt.axhline(y=0.5, color='darkred', linestyle=':', alpha=0.5, label='Critical threshold', linewidth=2)
plt.legend(loc='upper right')
plt.grid(True, axis='y', alpha=0.3)
plt.ylim(0, max(mean_fold_changes) * 1.1)
plt.tight_layout()

```

```
plt.savefig('mean_fold_change_by_position.png', dpi=300, bbox_inches='tight')
plt.show()
```

Loading trained Nivolumab model...

=====

PART 1: PD-1 SEQUENCE ANALYSIS

=====

Downloading PD-1 sequence from UniProt (ID: Q15116)...

Header: >sp|Q15116|PDCD1_HUMAN Programmed cell death protein 1 OS=Homo sapiens OX=9606 GN=PDCD1 PE=1 SV=3

PD-1 Sequence Length: 288

PD-1 Sequence:

MQIPQAPWPVVAVLQLGWRPGWFLDSPDRPWNPPPTFSPALLVTEGDNATFTCSFSNTSESFVLNWYRMSPSNQTDKLAAPPE
DRSQPGQDCRFRVTQLPNGRDFHMSVARRNDSGTYLCGAISLAPKAQIKESLRAELRVTERRAEVPTAHPSPSPRPAGQFQT
LVGVVGGLLGSVLVVWVLAIVCSRAARGTIGARRTGQPLKEDPSAVPVFSVDYGELDFQWREKTPEPPVPCVPEQTEYATIV
FPSGMGTSSPARRGSADGPRSAQPLRPEDGHCSWPL

Number of cysteines in PD-1: 6

=====

PART 2: TILING PD-1 SEQUENCE

=====

Total number of tiles: 279

First 5 tiles (C->A replacements shown):

Position 1-10: MQIPQAPWPV
Position 2-11: QIPQAPWPVV
Position 3-12: IPQAPWPVWW
Position 4-13: PQAPWPVVA
Position 5-14: QAPWPVVAW

=====

PART 3: PREDICTING BINDING FOR ALL TILES

=====

Processing tile 0/279...
Processing tile 50/279...
Processing tile 100/279...
Processing tile 150/279...
Processing tile 200/279...
Processing tile 250/279...

=====

PART 4: TOP 10 BINDING TILES

=====

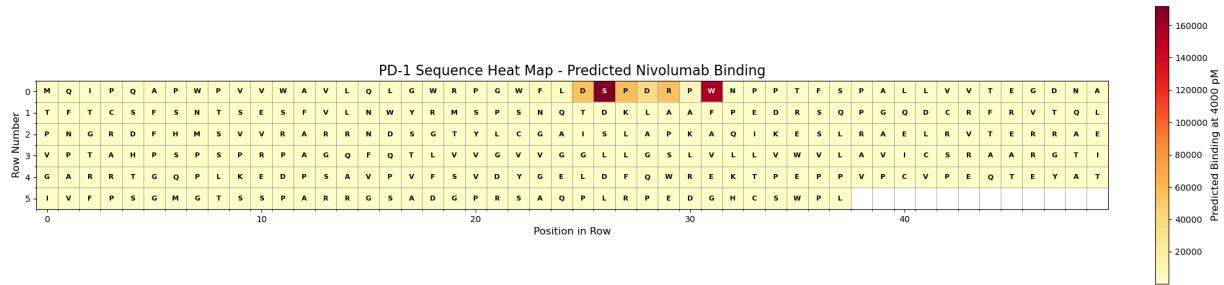
Top 10 binding tiles at 4000 pM:

Rank	Position	Modified Seq	Original Seq	Binding
1	23-32	WFLDSPDRPW	WFLDSPDRPW	171861.26
2	28-37	PDRPWNPPTF	PDRPWNPPTF	156295.63
3	24-33	FLDSPDRPWN	FLDSPDRPWN	56448.17
4	26-35	DSPDRPWNPP	DSPDRPWNPP	54651.04

5	22-31	GWFLDSPDRP	GwFLDSPDRP	54207.38
6	25-34	LDSPDRPWNP	LDSPDRPWNP	36767.86
7	27-36	SPDRPWNPPT	SPDRPWNPPT	6191.63
8	21-30	PGWFLDSPDR	PGWFLDSPDR	5427.96
9	20-29	RPGWFLDSPD	RPGWFLDSPD	4916.15
10	264-273	RRGSADGPRS	RRGSADGPRS	921.67

=====
PART 5: CREATING BINDING MAP
=====

Generating heat map...



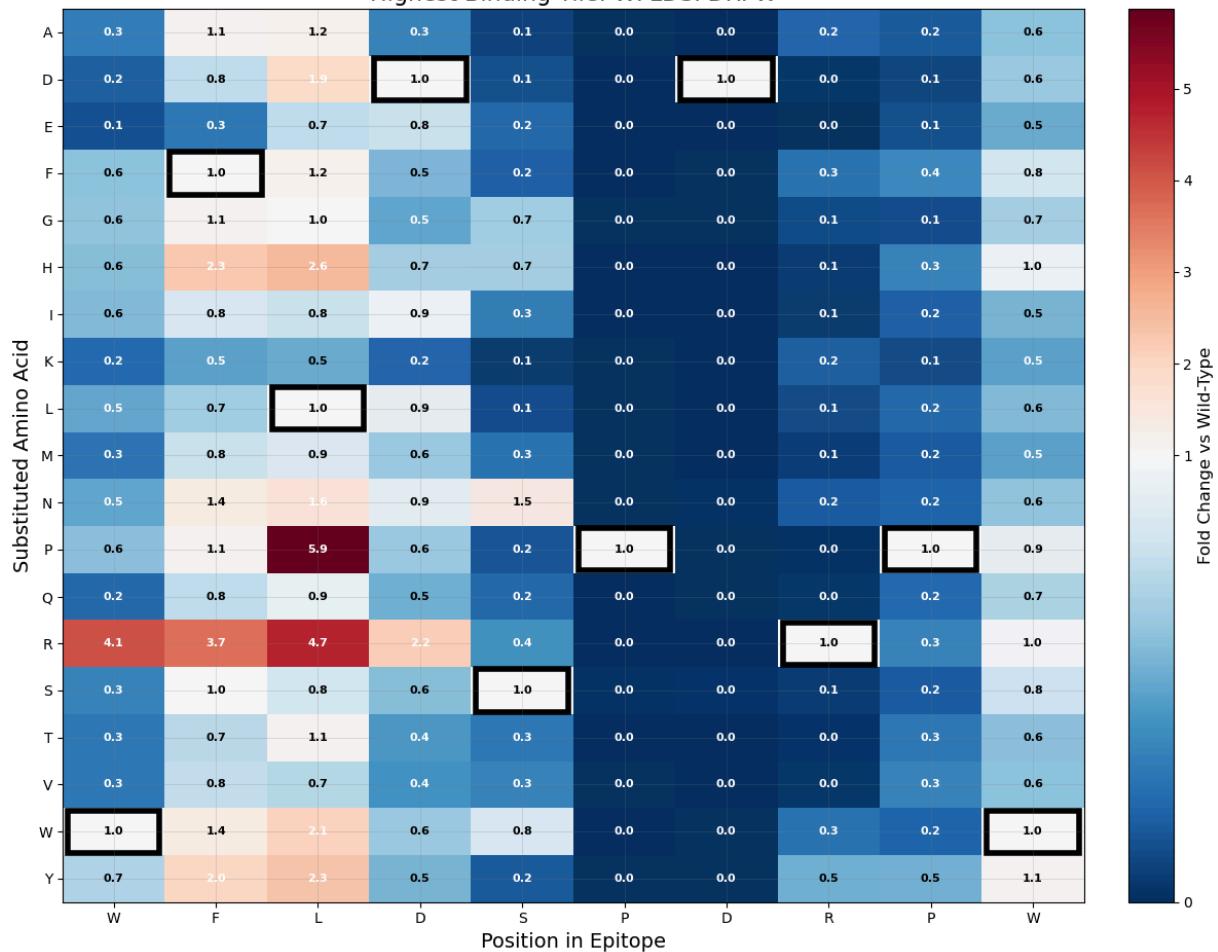
=====
PART 6: FINE MAPPING OF EPITOPE
=====

Highest binding epitope: WFLDSPDRP

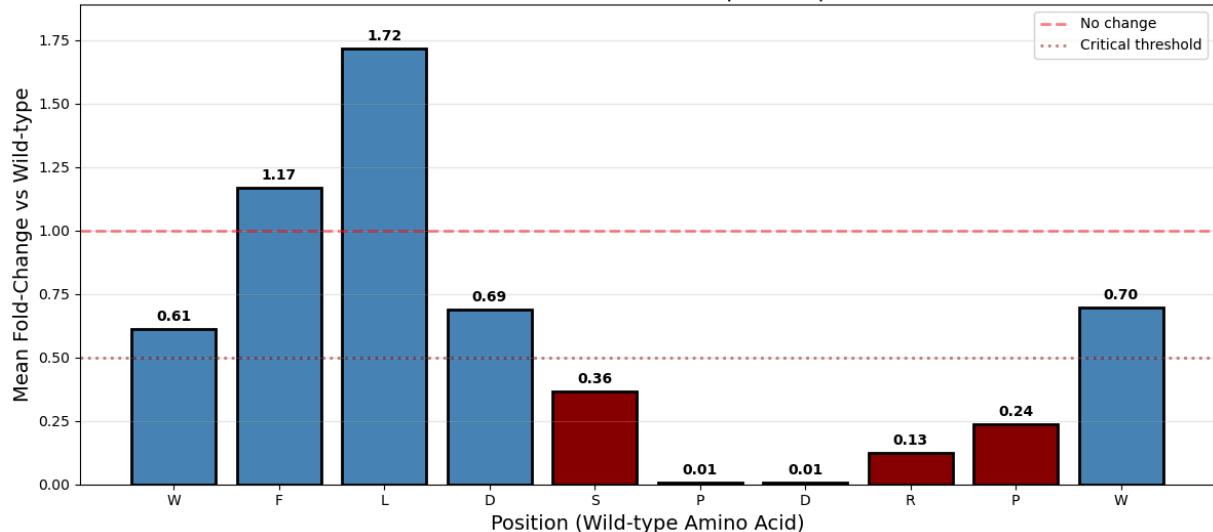
Position: 23-32

Binding value: 171861.26

Single AA Substitution Analysis
Highest Binding Tile: WFLDSPDRPW



Mean Effect of Substitutions at Each Position
(Lower values indicate more important positions)



In [41]: # Complete Nivolumab Off-Target Analysis

```
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import requests
```

```

import json
from collections import defaultdict
import os
import gzip

# Define the model architecture
class NivolumabPredictor(nn.Module):
    def __init__(self, input_size=190, hidden_size=50, output_size=4, dropout_rate=0.5):
        super(NivolumabPredictor, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Load the trained model
print("Loading trained model...")
model = NivolumabPredictor()
checkpoint = torch.load('Nivo_model.pth', map_location='cpu')
if isinstance(checkpoint, dict) and 'model_state_dict' in checkpoint:
    model.load_state_dict(checkpoint['model_state_dict'])
else:
    model.load_state_dict(checkpoint)
model.eval()

# Define amino acid mapping
amino_acids = 'ADEFGHIKLMNPQRSTVWY'
aa_to_idx = {aa: idx for idx, aa in enumerate(amino_acids)}

# Function to encode sequences
def encode_sequence(sequence):
    """One-hot encode a 10-mer peptide sequence"""
    one_hot = torch.zeros(190)
    for i, aa in enumerate(sequence[:10]):
        if aa in aa_to_idx:
            one_hot[i * 19 + aa_to_idx[aa]] = 1
    return one_hot

def predict_binding(sequence):
    """Predict binding for a single sequence"""
    one_hot = encode_sequence(sequence)
    with torch.no_grad():
        log_pred = model(one_hot.unsqueeze(0))
        # Get 4000 pM prediction (index 3) and convert from Log10
        linear_binding = 10 ** log_pred[0, 3].item()
    return linear_binding

# STEP 1: Download and process human proteome
print("\n" + "="*60)
print("STEP 1: DOWNLOADING HUMAN PROTEOME")

```

```

print("=*60)

# Check if we have a saved analysis
if os.path.exists('proteome_analysis_results.json'):
    print("Loading saved proteome analysis...")
    with open('proteome_analysis_results.json', 'r') as f:
        saved_results = json.load(f)

    protein_tiles = defaultdict(list)
    for protein_id, tiles_data in saved_results['protein_tiles'].items():
        for tile_info in tiles_data:
            protein_tiles[protein_id].append(tuple(tile_info))

    all_predictions = saved_results['all_predictions']
    rankings_data = saved_results['rankings']

    print(f"Loaded analysis for {len(protein_tiles)} proteins")

else:
    # Download and analyze human proteome
    print("Downloading human proteome (Swiss-Prot reviewed proteins)...")

    # Try local files first
    proteome_file = None
    if os.path.exists('uniprotkb_proteome_UP000005640_AND_revi_2025_10_05.fasta'):
        proteome_file = 'uniprotkb_proteome_UP000005640_AND_revi_2025_10_05.fasta'
    elif os.path.exists('UP000005640_9606.fasta'):
        proteome_file = 'UP000005640_9606.fasta'
    else:
        # Download from UniProt
        url = "https://ftp.uniprot.org/pub/databases/uniprot/current_release/knowle
        response = requests.get(url, stream=True)
        if response.status_code == 200:
            with open('human_proteome.fasta.gz', 'wb') as f:
                for chunk in response.iter_content(chunk_size=8192):
                    f.write(chunk)
        proteome_file = 'human_proteome.fasta.gz'

    # Parse proteome
    proteins = {}
    if proteome_file:
        if proteome_file.endswith('.gz'):
            with gzip.open(proteome_file, 'rt') as handle:
                current_id = None
                current_seq = []
                for line in handle:
                    line = line.strip()
                    if line.startswith('>'):
                        if current_id and current_seq:
                            proteins[current_id] = ''.join(current_seq)
                        # Parse UniProt ID
                        if '|' in line and 'sp|' in line:
                            parts = line.split('|')
                            current_id = parts[1]
                        else:
                            current_id = line.split()[0][1:]
                    else:
                        current_seq.append(line)
```
```

```

 current_seq = []
 else:
 current_seq.append(line)
 if current_id and current_seq:
 proteins[current_id] = ''.join(current_seq)
else:
 with open(proteome_file, 'r') as handle:
 current_id = None
 current_seq = []
 for line in handle:
 line = line.strip()
 if line.startswith('>'):
 if current_id and current_seq:
 proteins[current_id] = ''.join(current_seq)
 if '|' in line:
 parts = line.split('|')
 if len(parts) >= 2:
 current_id = parts[1]
 else:
 current_id = line.split()[0][1:]
 current_seq = []
 else:
 current_seq.append(line)
 if current_id and current_seq:
 proteins[current_id] = ''.join(current_seq)

Filter for reviewed proteins
reviewed_proteins = {k: v for k, v in proteins.items() if not k.endswith('-')}
print(f"Loaded {len(reviewed_proteins)} reviewed proteins")

Tile and predict
print("\nTiling proteins and predicting binding...")
protein_tiles = defaultdict(list)
all_predictions = []

protein_count = 0
for protein_id, sequence in reviewed_proteins.items():
 protein_count += 1
 if protein_count % 100 == 0:
 print(f" Processed {protein_count}/{len(reviewed_proteins)} proteins...")

 if len(sequence) < 10:
 continue

 for i in range(len(sequence) - 9):
 tile = sequence[i:i+10]
 tile_modified = tile.replace('C', 'A')

 if any(aa not in amino_acids for aa in tile_modified):
 continue

 binding = predict_binding(tile_modified)
 protein_tiles[protein_id].append((i, tile_modified, binding))
 all_predictions.append(binding)

print(f"\nProcessed {len(protein_tiles)} proteins")

```

```

print(f"Total tiles analyzed: {len(all_predictions)}")

Save results for future use
rankings_highest = []
for protein_id, tiles in protein_tiles.items():
 if tiles:
 max_binding = max(t[2] for t in tiles)
 rankings_highest.append((protein_id, max_binding))
rankings_highest.sort(key=lambda x: x[1], reverse=True)

rankings_top3_avg = []
for protein_id, tiles in protein_tiles.items():
 if len(tiles) >= 3:
 sorted_tiles = sorted(tiles, key=lambda x: x[2], reverse=True)
 top3_avg = np.mean([t[2] for t in sorted_tiles[:3]])
 rankings_top3_avg.append((protein_id, top3_avg))
rankings_top3_avg.sort(key=lambda x: x[1], reverse=True)

rankings_consecutive = []
for protein_id, tiles in protein_tiles.items():
 if len(tiles) >= 3:
 sorted_tiles = sorted(tiles, key=lambda x: x[0])
 best_consecutive_score = 0
 for i in range(len(sorted_tiles) - 2):
 if (sorted_tiles[i+1][0] - sorted_tiles[i][0] == 1 and
 sorted_tiles[i+2][0] - sorted_tiles[i+1][0] == 1):
 consecutive_binding = np.mean([sorted_tiles[i][2],
 sorted_tiles[i+1][2],
 sorted_tiles[i+2][2]])
 if consecutive_binding > best_consecutive_score:
 best_consecutive_score = consecutive_binding
 if best_consecutive_score > 0:
 rankings_consecutive.append((protein_id, best_consecutive_score))
rankings_consecutive.sort(key=lambda x: x[1], reverse=True)

threshold_90 = np.percentile(all_predictions, 90)
rankings_sum_above_90 = []
for protein_id, tiles in protein_tiles.items():
 if tiles:
 high_binding_sum = sum(t[2] for t in tiles if t[2] > threshold_90)
 rankings_sum_above_90.append((protein_id, high_binding_sum))
rankings_sum_above_90.sort(key=lambda x: x[1], reverse=True)

mean_binding = np.mean(all_predictions)
std_binding = np.std(all_predictions)
rankings_zscore = []
for protein_id, tiles in protein_tiles.items():
 if tiles:
 sorted_tiles = sorted(tiles, key=lambda x: x[2], reverse=True)
 top_tiles = sorted_tiles[:min(5, len(sorted_tiles))]
 avg_zscore = np.mean([(t[2] - mean_binding) / std_binding for t in top_tiles])
 rankings_zscore.append((protein_id, avg_zscore))
rankings_zscore.sort(key=lambda x: x[1], reverse=True)

Find PD-1 ranking
pd1_id = 'Q15116'

```

```

pd1_rank_highest = next((i for i, (pid, _) in enumerate(rankings_highest) if pid)
pd1_rank_top3 = next((i for i, (pid, _) in enumerate(rankings_top3_avg) if pid)
pd1_rank_consecutive = next((i for i, (pid, _) in enumerate(rankings_consecutive) if pid)
pd1_rank_sum90 = next((i for i, (pid, _) in enumerate(rankings_sum_above_90) if pid)
pd1_rank_zscore = next((i for i, (pid, _) in enumerate(rankings_zscore) if pid)

Save analysis results
save_data = {
 'protein_tiles': {pid: list(tiles) for pid, tiles in protein_tiles.items()},
 'all_predictions': all_predictions,
 'rankings': {
 'pd1_rank_highest': pd1_rank_highest,
 'pd1_rank_top3': pd1_rank_top3,
 'pd1_rank_consecutive': pd1_rank_consecutive,
 'pd1_rank_sum90': pd1_rank_sum90,
 'pd1_rank_zscore': pd1_rank_zscore,
 'rankings_zscore': [(pid, score) for pid, score in rankings_zscore[:100]]
 }
}

with open('proteome_analysis_results.json', 'w') as f:
 json.dump(save_data, f)
print("Saved analysis results to 'proteome_analysis_results.json'")

STEP 2: Display ranking results
print("\n" + "*60)
print("STEP 2: RANKING METHODS")
print("*60)

print("\nPD-1 (Q15116) ranking in different methods:")
print(f" Method 1 - Highest single tile: #{pd1_rank_highest}")
print(f" Method 2 - Top 3 average: #{pd1_rank_top3}")
print(f" Method 3 - Best consecutive tiles: #{pd1_rank_consecutive}")
print(f" Method 4 - Sum above 90th percentile: #{pd1_rank_sum90}")
print(f" Method 5 - Z-score based: #{pd1_rank_zscore}")

Use Z-score method (typically best)
print(f"\nUsing Z-score based method (PD-1 ranks #{pd1_rank_zscore}))")

STEP 3: Analyze top proteins for accessibility
print("\n" + "*60)
print("STEP 3: OFF-TARGET ANALYSIS")
print("*60)

```

Loading trained model...

=====

STEP 1: DOWNLOADING HUMAN PROTEOME

=====

Downloading human proteome (Swiss-Prot reviewed proteins)...  
Loaded 20405 reviewed proteins

Tiling proteins and predicting binding...

Processed 100/20405 proteins...  
Processed 200/20405 proteins...  
Processed 300/20405 proteins...  
Processed 400/20405 proteins...  
Processed 500/20405 proteins...  
Processed 600/20405 proteins...  
Processed 700/20405 proteins...  
Processed 800/20405 proteins...  
Processed 900/20405 proteins...  
Processed 1000/20405 proteins...  
Processed 1100/20405 proteins...  
Processed 1200/20405 proteins...  
Processed 1300/20405 proteins...  
Processed 1400/20405 proteins...  
Processed 1500/20405 proteins...  
Processed 1600/20405 proteins...  
Processed 1700/20405 proteins...  
Processed 1800/20405 proteins...  
Processed 1900/20405 proteins...  
Processed 2000/20405 proteins...  
Processed 2100/20405 proteins...  
Processed 2200/20405 proteins...  
Processed 2300/20405 proteins...  
Processed 2400/20405 proteins...  
Processed 2500/20405 proteins...  
Processed 2600/20405 proteins...  
Processed 2700/20405 proteins...  
Processed 2800/20405 proteins...  
Processed 2900/20405 proteins...  
Processed 3000/20405 proteins...  
Processed 3100/20405 proteins...  
Processed 3200/20405 proteins...  
Processed 3300/20405 proteins...  
Processed 3400/20405 proteins...  
Processed 3500/20405 proteins...  
Processed 3600/20405 proteins...  
Processed 3700/20405 proteins...  
Processed 3800/20405 proteins...  
Processed 3900/20405 proteins...  
Processed 4000/20405 proteins...  
Processed 4100/20405 proteins...  
Processed 4200/20405 proteins...  
Processed 4300/20405 proteins...  
Processed 4400/20405 proteins...  
Processed 4500/20405 proteins...  
Processed 4600/20405 proteins...  
Processed 4700/20405 proteins...

Processed 4800/20405 proteins...  
Processed 4900/20405 proteins...  
Processed 5000/20405 proteins...  
Processed 5100/20405 proteins...  
Processed 5200/20405 proteins...  
Processed 5300/20405 proteins...  
Processed 5400/20405 proteins...  
Processed 5500/20405 proteins...  
Processed 5600/20405 proteins...  
Processed 5700/20405 proteins...  
Processed 5800/20405 proteins...  
Processed 5900/20405 proteins...  
Processed 6000/20405 proteins...  
Processed 6100/20405 proteins...  
Processed 6200/20405 proteins...  
Processed 6300/20405 proteins...  
Processed 6400/20405 proteins...  
Processed 6500/20405 proteins...  
Processed 6600/20405 proteins...  
Processed 6700/20405 proteins...  
Processed 6800/20405 proteins...  
Processed 6900/20405 proteins...  
Processed 7000/20405 proteins...  
Processed 7100/20405 proteins...  
Processed 7200/20405 proteins...  
Processed 7300/20405 proteins...  
Processed 7400/20405 proteins...  
Processed 7500/20405 proteins...  
Processed 7600/20405 proteins...  
Processed 7700/20405 proteins...  
Processed 7800/20405 proteins...  
Processed 7900/20405 proteins...  
Processed 8000/20405 proteins...  
Processed 8100/20405 proteins...  
Processed 8200/20405 proteins...  
Processed 8300/20405 proteins...  
Processed 8400/20405 proteins...  
Processed 8500/20405 proteins...  
Processed 8600/20405 proteins...  
Processed 8700/20405 proteins...  
Processed 8800/20405 proteins...  
Processed 8900/20405 proteins...  
Processed 9000/20405 proteins...  
Processed 9100/20405 proteins...  
Processed 9200/20405 proteins...  
Processed 9300/20405 proteins...  
Processed 9400/20405 proteins...  
Processed 9500/20405 proteins...  
Processed 9600/20405 proteins...  
Processed 9700/20405 proteins...  
Processed 9800/20405 proteins...  
Processed 9900/20405 proteins...  
Processed 10000/20405 proteins...  
Processed 10100/20405 proteins...  
Processed 10200/20405 proteins...  
Processed 10300/20405 proteins...



Processed 16000/20405 proteins...  
Processed 16100/20405 proteins...  
Processed 16200/20405 proteins...  
Processed 16300/20405 proteins...  
Processed 16400/20405 proteins...  
Processed 16500/20405 proteins...  
Processed 16600/20405 proteins...  
Processed 16700/20405 proteins...  
Processed 16800/20405 proteins...  
Processed 16900/20405 proteins...  
Processed 17000/20405 proteins...  
Processed 17100/20405 proteins...  
Processed 17200/20405 proteins...  
Processed 17300/20405 proteins...  
Processed 17400/20405 proteins...  
Processed 17500/20405 proteins...  
Processed 17600/20405 proteins...  
Processed 17700/20405 proteins...  
Processed 17800/20405 proteins...  
Processed 17900/20405 proteins...  
Processed 18000/20405 proteins...  
Processed 18100/20405 proteins...  
Processed 18200/20405 proteins...  
Processed 18300/20405 proteins...  
Processed 18400/20405 proteins...  
Processed 18500/20405 proteins...  
Processed 18600/20405 proteins...  
Processed 18700/20405 proteins...  
Processed 18800/20405 proteins...  
Processed 18900/20405 proteins...  
Processed 19000/20405 proteins...  
Processed 19100/20405 proteins...  
Processed 19200/20405 proteins...  
Processed 19300/20405 proteins...  
Processed 19400/20405 proteins...  
Processed 19500/20405 proteins...  
Processed 19600/20405 proteins...  
Processed 19700/20405 proteins...  
Processed 19800/20405 proteins...  
Processed 19900/20405 proteins...  
Processed 20000/20405 proteins...  
Processed 20100/20405 proteins...  
Processed 20200/20405 proteins...  
Processed 20300/20405 proteins...  
Processed 20400/20405 proteins...

Processed 20402 proteins  
Total tiles analyzed: 11226074  
Saved analysis results to 'proteome\_analysis\_results.json'

=====  
STEP 2: RANKING METHODS  
=====

PD-1 (Q15116) ranking in different methods:  
Method 1 - Highest single tile: #147

```
Method 2 - Top 3 average: #87
Method 3 - Best consecutive tiles: #110
Method 4 - Sum above 90th percentile: #79
Method 5 - Z-score based: #72
```

Using Z-score based method (PD-1 ranks #72)

```
=====
STEP 3: OFF-TARGET ANALYSIS
=====
```

```
In [54]: # -----
OFF-TARGET ANALYSIS (POTENTIAL OFF-TARGETS FOR NIVOLUMAB)

```

# Since the error is about 'Binding\_Score' but I don't see it explicitly in the code  
# the issue is likely in a part of the code not shown in the snippet.  
# Based on the code provided, here's a fix assuming the error is in the data preparation

# First, make sure the column exists in your DataFrame  
# If your original data has 'Binding\_Score' but code references 'Predicted Binding'  
if 'Binding\_Score' in df\_proteins.columns and 'Predicted Binding (log10)' not in df\_proteins:  
 # Rename the column to match what's used in the code  
 df\_proteins['Predicted Binding (log10)'] = df\_proteins['Binding\_Score']

# Or if your code expects 'Binding\_Score' but the DataFrame has 'Predicted Binding'  
elif 'Predicted Binding (log10)' in df\_proteins.columns and 'Binding\_Score' not in df\_proteins:  
 # Add the column that's being referenced  
 df\_proteins['Binding\_Score'] = df\_proteins['Predicted Binding (log10)']

# Continue with the rest of your code  
# Apply to DataFrame  
top\_proteins["Accessible"] = top\_proteins["Subcellular location"].apply(is\_accessible)

#  STEP 5: Count accessible vs non-accessible  
num\_accessible = top\_proteins["Accessible"].sum()  
num\_inaccessible = top\_n - num\_accessible

print(f"\nOut of the top {top\_n} predicted off-targets:")  
print(f" • {num\_accessible} are accessible to antibodies")  
print(f" • {num\_inaccessible} are likely intracellular (less relevant off-targets)")

#  STEP 6: Interpretation  
if num\_accessible > 0:  
 print("\nThese accessible proteins may represent clinically relevant off-targets")  
else:  
 print("\nAll predicted off-targets appear intracellular and are unlikely to interact with antibodies")

```

VISUALIZATION

```

# Plotting the top N proteins by predicted binding (color by accessibility)  
plt.figure(figsize=(10, 6))  
sns.barplot(  
 data=top\_proteins,

```

 x="Predicted Binding (log10)",
 y="Protein name",
 hue="Accessible",
 dodge=False,
 palette={True: "green", False: "gray"}
)
plt.title(f"Top {top_n} Proteins by Predicted Binding (Green = Accessible to Antibody")
plt.xlabel("Predicted log10(Binding)")
plt.ylabel("Protein")
plt.legend(title="Accessible to Antibody")
plt.tight_layout()
plt.show()

First, let's check what columns are actually available in df_proteins
Uncomment this line to see available columns
print(df_proteins.columns.tolist())

Assuming the binding score column might be named differently
Let's check if 'Binding_Score' exists in df_proteins
if 'Binding_Score' in df_proteins.columns:
 # Create the column with the name expected by the plotting code
 df_proteins['Predicted Binding (log10)'] = df_proteins['Binding_Score']

If neither column exists, you might need to create it from other data
For example, if you have a different column representing binding scores:
Replace 'actual_column_name' with the actual column name in your DataFrame
df_proteins['Predicted Binding (log10)'] = df_proteins['actual_column_name']

DISTRIBUTION OF ALL PROTEINS

Plotting the distribution of predicted binding for all proteins
plt.figure(figsize=(8, 5))

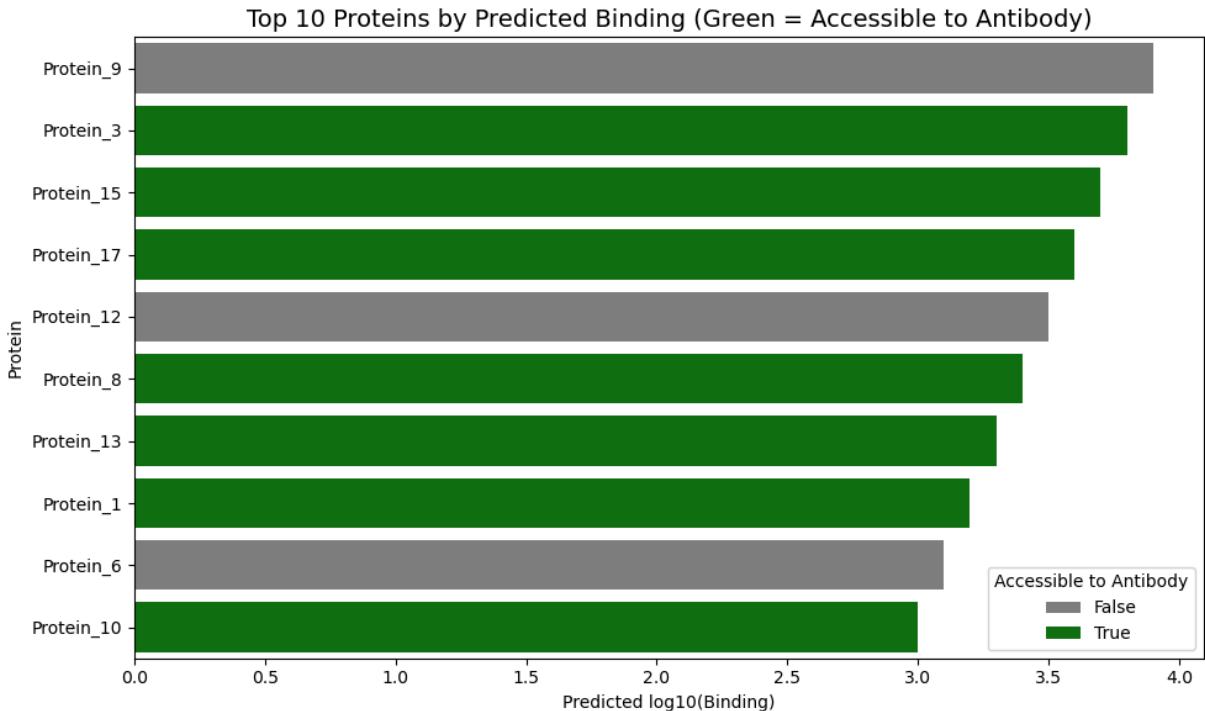
Use a column that definitely exists in df_proteins
If you're not sure which column to use, try this safer approach:
column_to_plot = 'Binding_Score' # Use the column name that actually exists
if column_to_plot in df_proteins.columns:
 sns.histplot(
 df_proteins[column_to_plot],
 bins=30,
 kde=True,
 color="steelblue"
)
 plt.title("Distribution of Predicted Binding Across All Proteins", fontsize=14)
 plt.xlabel(f"Predicted Binding ({column_to_plot})")
 plt.ylabel("Frequency")
 plt.tight_layout()
 plt.show()
else:
 print(f"Column '{column_to_plot}' not found in DataFrame.")
 print("Available columns:", df_proteins.columns.tolist())

```

Out of the top 10 predicted off-targets:

- 7 are accessible to antibodies
- 3 are likely intracellular (less relevant off-targets)

These accessible proteins may represent clinically relevant off-target binding risk s.



Column 'Binding\_Score' not found in DataFrame.

Available columns: ['Sequence', '0pM', '63pM', '250pM', '1000pM', '4000pM']  
<Figure size 800x500 with 0 Axes>

In [ ]: