POCKETMÓN
SBI-PYTHON PROJECT
DOCUMENTATION

AUTHORS

Marta García · marta.garcia38@estudiant.upf.edu
Karim Hamed · karim.hamed01@estudiant.upf.edu
Ivon Sánchez · ivon.sanchez01@estudiant.upf.edu

upf. Universitat Pompeu Fabra Barcelona

POCKÉTMON DOCUMENTATION

PACKAGE OVERVIEW

Pockétmon is a computational tool developed to predict ligand-binding pockets in protein structures using deep learning methodologies. It leverages a three-dimensional Convolutional Neural Network (3D-CNN) trained on protein-ligand complexes to identify regions of interest with high spatial and functional relevance.
Unlike classical approaches based on geometric descriptors, energy minimization, or evolutionary conservation, *Pockétmon* adopts a data-driven strategy, capable of learning implicit biochemical patterns from atomic configurations. This makes it particularly effective in identifying non-obvious binding regions and generalizing across structurally diverse protein families.

KEY FEATURES AND ADVANTAGES

Deep Learning-Powered Predictions

Pockétmon employs a 3D-CNN capable of modeling complex spatial relationships between atoms within a protein structure. This allows the tool to detect subtle features often missed by rule-based or empirical scoring methods.

Residue-Level Annotations

In addition to generating voxel-based predictions in the form of pseudo-atom PDB files, *Pockétmon* maps predicted pockets back to protein residues. The output includes a curated list of candidate binding site residues, facilitating biological interpretation and experimental validation.

Standalone and Efficient

The tool is designed for local execution, with no reliance on external APIs or databases. It supports both CPU and GPU computation (including CUDA and Apple MPS), allowing efficient processing of large datasets or high-throughput screening pipelines.

Tunable Trust Threshold

Users may specify a confidence threshold (--trust) to control the model's sensitivity in classifying voxels. Lower thresholds increase detection breadth (useful for exploratory analysis), while higher thresholds yield more conservative and specific predictions.

Pockétmon can support a wide range of structural bioinformatics tasks, including:

- Identification of druggable sites in novel or under-characterized proteins
- Prioritization of residues for mutagenesis or functional studies
- Ligand placement and docking pre-processing
- High-throughput annotation of proteome-wide structural datasets

By combining machine learning robustness with interpretability and ease of use, *Pockétmon* provides a reliable and scalable solution for protein pocket prediction in both academic and applied research context.

INSTALLATION

To ensure optimal compatibility and performance, it is strongly recommended to use Python 3.10, especially when running on systems utilizing Apple's Metal Performance Shaders (MPS) or when working with PyTorch-based environments.

Recommended: Use a Python 3.10 environment for best compatibility.

1. Clone the repository

```
git clone https://github.com/martagarnt/pocketmon.git
cd pocketmon
```

2. Create a virtual environment

If you're using Python 3.12+, you might encounter errors since some dependencies are not yet compatible with the latest Python versions. In that case, we suggest creating a clean Python 3.10 environment using Conda:

2.1. Create a virtual environment using Conda:

```
conda create -n pocketmon-env python=3.10
conda activate pocketmon-env
```

2.2. Create a regular virtual environment

In the case you have Python 3.10, no Conda environment needs to be created; just make sure the version is specified when calling python commands (in case you have serveral versions downloaded).

```
python3.10 -m venv venv
source venv/bin/activate
```

3. Install dependencies

```
pip install -r requirements.txt
```

4. Install the CLI tool

```
pip install .
```

After this, the pocketmon command will be globally accessible in the terminal.

VERIFYING INSTALLATION

To test if the installation was successful, run:

```
python3.10 -c "from pocketmon import predict;
predict.print_banner('install'); print('✓ Pockétmon installed
successfully and ready to catch some pockets\!')"
```

USAGE

```
pocketmon [options] --input <pdb_file_path>
```

OPTIONS

- --model <model_path>: Path to the trained model weights (.pt file) [default: best_model_refined.pt]
- --output <output_basename>: Base name for output files [default: predicted_pocket]
- --trust <float>: Trust threshold for voxel classification (0.0–1.0) [default: 0.5]
- -v, --verbose: Enable verbose output
- -h, --help: Show help message and exit

REQUIRED ARGUMENTS

- --input <pdb_file_path>: Path to the input PDB file

OUTPUT

- A PDB file containing voxel pseudo-atoms representing the predicted pocket:
  - <output_basename>_predicted_pocket.pdb
- A text file listing predicted residues:
  - <output_basename>_predicted_residues.txt
  - Format: RESNAME RESID CHAIN (e.g., LEU 6 A)

TRUST THRESHOLD EXPLAINED

The trust threshold sets how confident the model must be before classifying a voxel as part of the pocket:

- **Low values (e.g. 0.3)** → More voxels included, more recall
- **High values (e.g. 0.8)** → Fewer voxels, higher precision
- **Default (0.5)** → Balanced tradeoff

Tune this parameter to your needs using --trust.

EXAMPLE USAGE

```
pocketmon --input myprotein.pdb
pocketmon -v --input 2ay2.pdb --model best_model_refined.pt --
output result --trust 0.6
```

DOCUMENTATION

DATA_PREPROCESSING.IPYNB

Function collect_pdb_pairs

```
def collect_pdb_pairs(dataset_root):
```

Description:

This function navigates a dataset directory where each subdirectory contains at least one protein and one pocket PDB file. It looks for files ending in _protein.pdb and pocket.pdb, and stores the first match of each per subdirectory. It returns two lists of aligned file paths for protein and pocket structures, which are useful for downstream analysis such as training a deep learning model for binding site prediction.

Input: dataset_root (str): Path to the root folder containing subdirectories, each corresponding to a sample or structure group.

Output:

- Tuple [List[str], List[str]]:
- Two lists of strings:
  - Full paths to _protein.pdb files.
  - Full paths to _pocket.pdb files.
    Each index i in both lists correspond to a matched pair: protein_paths[i] ↔ pocket_paths[i].

Use case:
Useful in preprocessing pipelines where models require aligned input of protein structures and their known binding pockets. For example, used before voxelization or data augmentation for a CNN-based binding pocket predictor.

Remarks:
- Only the **first** file matching each pattern (*_protein.pdb, *_pocket.pdb) is selected in each subdirectory.
- Ignores non-directory files in the root.
- Assumes consistent naming conventions across the dataset.

```
voxelize_structure(pdb_path, origin=None, grid_size=32,
voxel_size=1.0, channels=['C', 'N', 'O', 'S'],
return_origin=False)
```

Description:
This function converts atomic coordinates from a PDB file into a voxelized 3D grid with separate channels for specified atom types. It processes a PDB structure and maps it into a 4D voxel grid (channels x D x H x W) where each voxel counts the atoms of a given type that fall within its space. Optionally, you can specify the grid's origin or let the function automatically center the grid on the molecule.

Input:

- pdb_path (str): Path to the input .pdb file containing the protein structure.
- origin (np.ndarray or None): Center of the grid in 3D space. If None, it's computed automatically to center the structure.
- grid_size (int): Number of voxels per axis (default: 32).
- voxel_size (float): Physical size of each voxel in Ångströms (default: 1.0).
- channels (list of str): Atom types to map into the grid (e.g., ['C', 'N', 'O', 'S']).
- return_origin (bool): If True, the computed/used origin is returned alongside the voxel grid.

Output:

- grid (np.ndarray): A 4D float array of shape (len(channels), grid_size, grid_size, grid_size) representing the spatial distribution of atoms per type.
- origin (np.ndarray, optional): A 3D vector representing the origin used for voxelization (only returned if return_origin=True).

Use Case:
Converts raw structural data into a format suitable for 3D convolutional neural networks (CNNs). Useful for tasks like protein binding pocket prediction, structure classification, or ligand docking.

Remarks:

- Atom types not included in channels are ignored.
- Atom coordinates are binned into voxels using simple integer floor division.
- Multiple atoms may fall into the same voxel, incrementing the voxel value.
- If the origin is automatically calculated, the grid is centered on the molecule.
- Edge cases where atoms fall just outside the grid bounds are ignored silently.

- This voxelization does not apply any smoothing or spatial decay (like Gaussian kernels), which may be desirable for certain ML tasks.

```
def generate_label_grid(pocket_pdb_path, origin, grid_size=32,
voxel_size=1.0)
```

Description:
This function enerates a 3D binary label grid from a pocket PDB file using a predefined voxelization origin. It creates a voxel grid (3D NumPy array) where each voxel indicates whether it contains part of the binding pocket. It uses the same grid parameters as the input protein voxelization (same origin, size, and voxel scale) to ensure spatial alignment between input data and target labels.

Input:
- pocket_pdb_path (str): Path to the .pdb file containing atoms of the binding pocket.
- origin (np.ndarray): Origin of the voxel grid used to align coordinates, same as in voxelize_structure().
- grid_size (int): Number of voxels per axis (default: 32).
- voxel_size (float): Size of each voxel in Ångströms (default: 1.0).

Output:
- label_grid (np.ndarray): A 3D array of shape (grid_size, grid_size, grid_size), where 1 indicates presence of pocket atoms and 0 indicates absence.

Use Case:
Used to generate ground truth labels for supervised learning, particularly in 3D CNN models that learn to identify or predict protein binding pockets.

Remarks:
- Each atom in the pocket structure is assigned to a single voxel using its coordinates.
- Atom positions outside the grid boundaries are ignored.
- The label grid aligns spatially with voxelized protein inputs to support voxel-wise learning.
- Unlike the input voxel grid, this label grid is binary and single-channeled—ideal for segmentation tasks.

```
class PocketGridDataset(Dataset):
```

Description:
This class enables training of deep learning models (e.g., 3D CNNs) by wrapping a set of matched protein and pocket structures. It voxelizes each protein into a multi-channel input grid and each pocket into a single-channel binary label grid, keeping both spatially aligned via a shared origin.

Input:

- protein_paths (List[str]): File paths to protein .pdb files.
- pocket_paths (List[str]): Corresponding file paths to pocket .pdb files.
- grid_size (int): Size of the voxel grid along each axis (default: 32).
- voxel_size (float): Physical size of each voxel in Ångströms (default: 1.0).

Output (from __getitem__):

- X (torch.Tensor): 4D tensor of shape (channels, grid_size, grid_size, grid_size) representing the voxelized protein.
- Y (torch.Tensor): 4D tensor of shape (1, grid_size, grid_size, grid_size) representing the binary label grid for the pocket.

*Output of __len__ ; it returns the number of samples, defined by the length of protein_paths.

Use Case:
Designed for 3D CNN training pipelines in tasks such as binding site prediction, ligandability classification, or structure segmentation.

Remarks:

- Assumes 1-to-1 matching between entries in protein_paths and pocket_paths.
- Automatically calls voxelize_structure() and generate_label_grid() for each index.
- Converts output to torch.Tensor and ensures labels have an added channel dimension (unsqueeze(0)).

```
class VoxelNPYDataset(torch.utils.data.Dataset):
```

Description:
This dataset is designed for cases where voxelized protein and pocket data have been precomputed and saved as .npyfiles (e.g., X_0.npy, Y_0.npy, etc.). It allows quick and efficient loading during model training without recomputing voxel grids on-the-fly.

Input:
- voxel_dir (str): Path to the directory containing the preprocessed .npy files.
- total_samples (int): Total number of samples expected in the directory (used to define __len__).

Output (from __getitem__):
- X (torch.Tensor): Tensor loaded from X_{idx}.npy, representing the voxelized protein grid (typically shape: (channels, D, H, W)).
- Y (torch.Tensor): Tensor loaded from Y_{idx}.npy, representing the binary label grid (typically shape: (1, D, H, W)).

*same case as the previous class; __len__ output returns the total number of samples.

Use Case:
Best suited for workflows where voxelization has already been done as a preprocessing step, allowing fast iteration and training.

Remarks:
- Filenames must follow the exact pattern X_{idx}.npy and Y_{idx}.npy (e.g., X_0.npy, Y_0.npy).
- Assumes all .npy files are correctly formatted and located in voxel_dir.
- Useful for deploying training pipelines on datasets that are too large.

MODEL_TRAINING.IPYNB

Class Pocket3DCNN

```
class Pocket3DCNN(nn.Module):
    def __init__(self, in_channels=4):
[…]
    def forward(self, x):
[…]
```

Description:
Pocket3DCNN processes 3D voxelized protein structures and predicts, for each voxel, whether it belongs to a binding pocket. The architecture uses multiple convolutional layers with batch normalization and ReLU activations, ending in a 1-channel sigmoid output for binary segmentation. The model is designed to operate on inputs with multiple atom-type channels (e.g., C, N, O, S).

Input: in_channels (int): Number of input channels. Typically 4 for atom types ['C', 'N', 'O', 'S'].

Output (from forward): A tensor of shape (batch_size, 1, D, H, W), with voxel-wise probabilities indicating pocket presence (0–1, after sigmoid).

Use Case:
Ideal for learning to predict the location of ligand binding sites in proteins from voxelized structural data. Can be trained using voxel-wise binary cross-entropy loss (BCEWithLogitsLoss if sigmoid is removed, or BCELoss if kept).

Remarks:
- Uses nn.Sequential for a clean block-based definition.
- Final layer reduces to a single output channel with Sigmoid for voxel-wise binary classification.
- All conv layers have padding=1 to preserve spatial dimensions.
- You can adjust the number of layers, filters, or kernel sizes to balance model complexity and performance.

Function accuracy

```
accuracy(preds, labels)
```

Description:
Compares predicted probabilities (after applying a threshold of 0.5) to the binary label mask and calculates the proportion of correctly predicted voxels.

Input:
- preds (torch.Tensor): Prediction tensor with values in [0, 1].
- labels (torch.Tensor): Ground truth binary tensor.

Output: float: Accuracy value in the range [0.0, 1.0].

Remarks:
- Works on tensors of any shape.
- Assumes sigmoid activation has already been applied to preds.

Function precision

```
accuracy(preds, labels)
```

Description:
Precision is the ratio of correctly predicted pocket voxels to all predicted pocket voxels.

Input:
- preds (torch.Tensor): Prediction tensor.
- labels (torch.Tensor): Ground truth binary tensor.
- threshold (float): Classification threshold for predictions (default: 0.5).

Output: float: Precision score in [0.0, 1.0].

Remarks:
- Uses sklearn.metrics.precision_score.
- Converts tensors to flattened NumPy arrays.

Function recall

```
recall(preds, labels, threshold=0.5)
```

Definition:
Computes the voxel-wise recall score; it recalls the ratio of correctly predicted pocket voxels to all actual pocket voxels.

Input: Same as precision.

Output: float: Recall score in [0.0, 1.0]

Remarks:
- Uses sklearn.metrics.recall_score.
- Handles zero-division cases gracefully with zero_division=0.

Function f1_score

```
f1_score(prec, rec, eps=1e-8)
```

Description:
Calculates the F1 score given precision and recall. Harmonic mean of precision and recall. Useful for imbalanced classification.

Input:
- prec (float): Precision value.
- rec (float): Recall value.
- eps (float): Small epsilon to avoid division by zero (default: 1e-8).

Output:
- float: F1 score in [0.0, 1.0].

Function dice_loss

```
dice_loss(probs, target, smooth=1e-8)
```

Description:
The Dice coefficient measures overlap between prediction and ground truth. Dice loss is 1 - Dice score, encouraging overlap during training.

Input:
- probs (torch.Tensor): Predicted probabilities.
- target (torch.Tensor): Ground truth binary mask.
- smooth (float): Smoothing factor to prevent division by zero (default: 1e-8).

Output: float: Dice loss value.

Remarks:
- target is moved to probs's device.
- Often preferred for segmentation with imbalanced classes.

Function load_checkpoint

```
load_checkpoint(model, optimizer, filename='checkpoint.pth')
```

Description:
This function restores a saved PyTorch model and optimizer state from a checkpoint file. Useful for resuming training or evaluating a previously trained model.

Input:
- model (nn.Module): The model architecture to load weights into.
- optimizer (torch.optim.Optimizer): The optimizer used during training.
- filename (str): Path to the saved checkpoint file (default: 'checkpoint.pth').

Output:
- model: The model with loaded weights.
- optimizer: The optimizer with loaded state.epoch (int): The training epoch the checkpoint was saved at.
- loss (float): The loss recorded at the time of saving.

Use Case:
For resuming interrupted training sessions or evaluating saved models in consistent state.

Remarks:
- Ensure that model and optimizer have the same architecture and settings as during saving.
- Prints the loaded epoch for user feedback.

Function save_checkpoint

```
save_checkpoint(model, optimizer, epoch, loss,
filename='checkpoint.pth')
```

Description:
Captures the training state at a given point for later resumption or evaluation. Encodes all essential components to restore training exactly as it was.

Input:
- model (nn.Module): The model to save.
- optimizer (torch.optim.Optimizer): The optimizer to save.
- epoch (int): The current training epoch.
- loss (float): The loss value at this epoch.
- filename (str): Destination filename for the checkpoint (default: 'checkpoint.pth').

Output: None (writes file to disk).

Use Case:
Useful for logging progress in long training sessions, implementing early stopping, or sharing pretrained models.

Remarks:
- Saves model and optimizer state_dicts, not full objects.
- Prints confirmation message with epoch number.

Description:
The main loop of the model_training.ipynb trains a 3D CNN on voxelized protein structures and evaluates its performance across training, validation, and test sets, with checkpointing and metric logging.

This script handles complete model training including:
- Resuming from checkpoints
- Epoch-wise training and validation
- Final test set evaluation
- Metric logging (loss, accuracy, precision, recall, F1)
- Model checkpointing and best model saving.

Input:
Assumes the following are defined prior:
- model, optimizer, loss_fn, device
- train_loader, val_loader, test_loader
- num_epochs, best_val_loss, metric logging lists

Output:
- Trained model weights saved as 'model.pt' if validation loss improves.
- Intermediate checkpoints saved to 'model_checkpoint.pth'.
- Printed logs per epoch for training and validation, and final performance on test set.

Use Case:
Trains a model to predict protein binding pockets from 3D voxel inputs. This script is production-ready for research, experimentation, or extension (e.g., hyperparameter tuning, early stopping, learning rate scheduling).

Remarks:
- Uses Dice loss, but you can easily switch to another like BCEWithLogitsLoss.
- Outputs are printed for every 100 training batches and after each epoch.
- Automatically resumes training from checkpoint if present.
- Saves best model based on lowest validation loss.
- All metrics are averaged over each dataset split.
- Be sure to define train_losses, train_accuracies, etc. as empty lists before training begins if you're logging metrics.
- To switch to multi-GPU training, you can wrap model with nn.DataParallel() or use DistributedDataParallel.

PREDICT.PY

```
class Pocket3DCNN(nn.Module):
    def __init__(self, in_channels=4):
[…]
    def forward(self, x):
[…]
```

Description:
The Pocket3DCNN class defines a three-dimensional convolutional neural network (3D-CNN) tailored for the task of identifying ligand-binding pockets within protein structures. It accepts voxelized protein input data and returns a probability map highlighting regions likely to represent binding sites.

This architecture is designed to learn spatial and chemical patterns from volumetric representations of proteins and leverages multiple convolutional layers to capture hierarchical features.

Constructor:

```
Pocket3DCNN(in_channels=4)
```

Parameter	Type	Description
in_channels	int	Number of input channels, typically corresponding to different atomic types (e.g., C, N, O, S). Default is 4.

Network Architecture:
The network consists of the following layers, organized in a `nn.Sequential` container:

1. Conv3D Layer 1
   - in_channels → 32 filters
   - Kernel size: 3×3×3, Padding: 1
   - Followed by: BatchNorm3D + ReLU
2. Conv3D Layer 2
   - 32 → 64 filters
   - Kernel size: 3×3×3, Padding: 1
   - Followed by: BatchNorm3D + ReLU
3. Conv3D Layer 3
   - 64 → 128 filters
   - Kernel size: 3×3×3, Padding: 1
   - Followed by: BatchNorm3D + ReLU

4. Conv3D Layer 4 (Output Layer)
    - 128 → 1 filter
    - Kernel size: 1×1×1
    - Followed by: Sigmoid activation

This produces a single-channel voxel grid of the same spatial dimensions as the input, where each voxel's value (between 0 and 1) represents the predicted likelihood of being part of a binding pocket.

Parameter	Type	Description
x	torch.Tensor	Input tensor of shape (batch_size, in_channels, D, H, W) representing voxelized protein structures.

Forward pass:

```
def forward(self,x):
     return self.model(x)
```

Returns	Type	Description
output	torch.Tensor	Output tensor of shape (batch_size, 1, D, H, W) containing voxel-wise pocket probability scores.

Use Case:
This model is optimized for protein binding site prediction by learning to associate structural patterns in 3D space with potential ligand interaction regions. It can be trained on protein-ligand complexes and later used to generalize predictions on unseen PDB structures.

Remarks:
The use of batch normalization and ReLU activation after each convolution layer enhances training stability and non-linearity.
The final Sigmoid function ensures output probabilities are bounded between 0 and 1, allowing for post-processing thresholding.

Function voxelize_structure

```
def voxelize_structure(pdb_path, origin=None, grid_size=32,
voxel_size=1.0, channels=['C', 'N', 'O', 'S']):
```

Description:
The voxelize_structure function processes a protein structure from a PDB file and converts it into a 3D voxel grid representation suitable for input into a convolutional neural network (CNN). This spatial encoding enables machine

learning models to learn local chemical environments from the atomic layout of the protein.

Parameters:

Parameter	Type	Description
`pdb_path`	`str`	Path to the input PDB file containing the protein structure.
`origin`	`np.ndarray or None`	Coordinates for the grid origin. If None, the centroid of all atom coordinates is used as the center of the voxel grid.
`grid_size`	`int`	Dimension of the cubic grid in voxels. The output grid will be of shape (C, grid_size, grid_size, grid_size). Default is 32.
`voxel_size`	`float`	Physical size (in Ångstroms) of each voxel. Defines the resolution of the grid. Default is 1.0.
`channels`	`list[str]`	List of atomic element types (e.g., 'C', 'N', 'O', 'S') to encode into distinct channels of the voxel grid.

Returns:

Return Value	Type	Description
`grid`	`np.ndarray`	4D array with dimensions (C, X, Y, Z) representing the voxelized protein structure, where each channel corresponds to an atomic type.
`origin`	`np.ndarray`	The 3D coordinates of the grid origin used during voxelization.
`structure`	`Bio.PDB.Structure`	Biopython structure object of the protein, used for downstream analysis and residue lookup.
`atom_residue_map`	`dict[tuple[int], Bio.PDB.Residue]`	Dictionary mapping voxel indices (x, y, z) to their corresponding Biopython Residue objects. This enables linking predicted voxel activations back to biological residues.

Use Case:

This function is a critical preprocessing step in workflows involving 3D CNNs for protein-ligand interaction prediction. It allows atomic-level data to be spatially structured, making it accessible for deep learning models that leverage spatial convolution.

Remarks:

- The accuracy of downstream predictions can be affected by the chosen voxel_size and grid_size.
- The atom_residue_map is essential for residue-level interpretation of voxel-level predictions.

Function save_predicted_pocket_to_pdb

```
def save_predicted_pocket_to_pdb(pred_grid, origin, voxel_size,
pdb_filename, threshold=0.5, header_name=None)
```

Description:

This function saves the predicted ligand-binding pocket as a 3D voxel-based PDB (Protein Data Bank) file. It converts voxels classified as part of the binding pocket (based on a confidence threshold) into pseudo-atoms in PDB format. Each pseudo-atom represents a voxel center within the predicted pocket volume.

This output can be visualized with molecular modeling software such as PyMOL or Chimera, enabling intuitive spatial inspection of predicted binding sites.

Parameters:

Parameter	Type	Description
`pred_grid`	`np.ndarray or torch.Tensor`	3D array of pocket prediction values (typically from a CNN). Values should be in the range [0, 1].
`origin`	`np.ndarray`	The 3D coordinates of the origin (minimum corner) of the voxel grid, used to convert voxel indices into Cartesian coordinates.
`voxel_size`	`float`	The physical size (in Ångströms) of a single voxel edge.
`pdb_filename`	`str`	The file path where the output PDB will be written.
`threshold`	`float, optional`	A probability cutoff above which voxels are considered part of the binding pocket. Default is 0.5.

`header_name`	`str, optional`	An optional protein name or identifier to be written in the PDB file header.

<u>Returns:</u>
This function does not return any value. It writes a .pdb file to disk containing pseudo-atoms for all voxels with prediction scores above the specified threshold.

<u>PDB Formatting Details:</u>
- Each qualifying voxel is written as a HETATM entry in the PDB file.
- The pseudo-atoms are labeled with a generic atom name X and residue UNK (unknown).
- Coordinates are computed by scaling voxel indices from the origin using the voxel size.
- The output includes a final END statement, conforming to standard PDB format requirements.

<u>Usage example:</u>

```
save_predicted_pocket_to_pdb(
    pred_grid=cnn_output,
    origin=np.array([10.0, 15.0, 5.0]),
    voxel_size=1.0,
    pdb_filename="protein_predicted_pocket.pdb",
    threshold=0.6,
    header_name="protein"
)
```

Function save_predicted_residues

```
def save_predicted_residues(pred_grid, origin, voxel_size,
residue_lookup, output_file, threshold=0.5)
```

<u>Description:</u>
This function extracts and saves the list of protein residues predicted to be part of a ligand-binding pocket. It maps high-confidence voxels (above the specified threshold) from the 3D prediction grid to corresponding residues using a voxel-to-residue lookup dictionary. The result is a residue-level annotation of the predicted pocket, saved in a human-readable text file.
This function complements the voxel-level .pdb output by providing biologically meaningful residue identifiers, which are easier to interpret and analyze.

__Parameters:__

Parameter	Type	Description
pred_grid	np.ndarray or torch.Tensor	3D array containing voxel prediction scores from the CNN.
origin	np.ndarray	The origin (minimum corner) of the voxel grid in 3D Cartesian coordinates.
voxel_size	float	The physical edge length of each voxel in Ångströms.
residue_lookup	dict	A dictionary mapping (x, y, z) voxel indices to their corresponding Bio.PDB.Residue objects. This is generated during voxelization.
output_file	str	Path to the .txt file to write the residue list.
threshold	float, optional	Probability threshold for voxel classification (default: 0.5). Voxels with predictions above this value are included.

__Returns:__
This function does not return a value. It writes a text file to disk with one line per residue, in the format:

```
# RESNAME RESID CHAIN
```

Example output:

```
LEU 6 A
ASN 8 A
LYS 10 A
```

__Implementation Details:__

- Converts PyTorch tensors to NumPy arrays if needed.
- Iterates through all voxels above the threshold.
- Uses the residue_lookup dictionary to associate each qualifying voxel with a unique residue.
- Avoids duplicate entries using a set to track seen residues.
- Sorts the residue list first by chain, then by residue number for consistency.

Usage example:

```
save_predicted_residues(
    pred_grid=cnn_output,
    origin=np.array([0.0, 0.0, 0.0]),
    voxel_size=1.0,
    residue_lookup=voxel_to_residue_dict,
    output_file="protein_predicted_residues.txt",
    threshold=0.6
)
```

Remarks:

- The output file is intended for downstream biological interpretation and validation.
- This residue-level output is especially useful when identifying potential ligand contact points or conducting mutagenesis studies.
- A well-constructed residue_lookup is essential for accurate mapping; it must come from the same voxelization logic used for prediction.

Function main

The main() function orchestrates the entire workflow of the Pockétmon prediction tool. It serves as the command-line interface (CLI) entry point, handling user input, model loading, data preprocessing, inference, and output generation. It is designed to be invoked directly when the program is run from the command line using the pocketmon command.

Workflow Overview:

1. **Help Screen Display:**
   If invoked with no arguments or with -h/--help, the function prints an ASCII art help screen summarizing usage, arguments, and example commands, then exits.
2. **Argument Parsing:**
   Uses Python's argparse library to parse the required and optional command-line arguments:
   - --input: path to the input PDB file (required)
   - --model: path to a .pt PyTorch model file (optional, defaults to best_model_refined.pt)
   - --output: base name for output files (default: predicted_pocket)
   - --trust: voxel confidence threshold for classification (float, 0.0–1.0, default: 0.5)
   - --verbose: enables additional console output for debugging and transparency
3. **Device Detection:**

Automatically selects the most optimal computing device:
   - GPU via CUDA (if available)
   - Apple MPS (if on macOS, with fallback warning)
   - CPU fallback otherwise
4. **Model Loading:**
   Initializes the 3D CNN architecture and loads the weights from the specified model file. The model is then moved to the selected device and set to evaluation mode.
5. **Input Preprocessing.**
   Invokes voxelize_structure() to parse the input PDB file and convert it into a voxel grid suitable for CNN input. Also retrieves the residue-to-voxel mapping.
6. **Prediction:**
7. Performs inference using the loaded model and converts raw sigmoid outputs into a binary mask based on the --trust threshold.
8. **Output Naming:**
   Dynamically generates filenames based on the input PDB file:
   - <protein_name>_predicted_pocket.pdb (voxel output)
   - <protein_name>_predicted_residues.txt (residue list)
9. **Saving Results:**
   - The .pdb file is saved using save_predicted_pocket_to_pdb().
   - The .txt residue list is saved using save_predicted_residues().
10. **Logging & Exit:**
   Prints output paths and key steps to the console, especially when --verbose is enabled. Ends execution gracefully.