epistatic_benchmarking

October 30, 2024

1 Benchmarking unsupervised and supervised models on the 4 site combinatorial library of TrpB

1. Johnston, K. E. et al. A combinatorially complete epistatic fitness landscape in an enzyme active site. Proceedings of the National Academy of Sciences 121, e2400439121 (2024).

The work in this notebook will look fundamentally similar to a subset of the work done by *Hsu et al.*, but in a single, readable, notebook. Additionally, we consider more than just OHE for AA embeddings.

We will probe slightly different factors than the original work as an "update" for those conclusions, on a epistatic benchmark. The goal is to do so in a way that highlights the integrated nature of AIDE and showcase how many different model types/methods can be accessed ina single, readable, notebook with the API in place.

Differences in this work: - Compare zero shot methods not in original paper to EV mutation: ESM2 and MSA transformer wild type marginal - Compare different embedding methods as oposed to just one hot encoding: ESM2 mean pooling over whole sequence, ESM2 mean pooling over only the 4 variable residues - Compare linear to a nonlinear top model - Conduct 5 fold CV not just for hyperparameter optimization, but also test set performance. Repeat with 20 random instantiations like the original paper.

Naming conventions:

`<zs_model>_<embedding>_<top_model>`

[ ]:
```
import os
import json

import pandas as pd
import numpy as np
import joblib

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="whitegrid")
sns.set_context("talk")

from sklearn.pipeline import Pipeline
from sklearn.compose import TransformedTargetRegressor
```

```
from sklearn.neural_network import MLPRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import KFold, RandomizedSearchCV, cross_validate
from sklearn.metrics import make_scorer
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import Ridge

from scipy.stats import kendalltau, spearmanr
from scipy.stats import loguniform, uniform

import aide_predict as ap
from aide_predict.utils.msa import MSAProcessing

import logging
logging.basicConfig(level=logging.INFO)
```

1.1 TOC

1.2 0. Acquiring the data

1. Download and extract the data from here. You want `data.zip`
2. Extract the data to a directory of your choice
3. Assign the global variable `RAW_DATA_DIR` to the extracted directory

[ ]:
```
RAW_DATA_DIR = os.path.join('.', 'data', 'epistatic', 'data')
```

1.2.1 0.1. Retrieve the assay labeled data

[ ]:
```
df = pd.read_csv(os.path.join(RAW_DATA_DIR, 'figure_data',␣
↪'4-site_merged_replicates', '20230827', 'four-site_summary_AA_data.csv'))
```

[ ]:
```
df = df[['AAs', 'avg_mu-bg/max']].rename(columns={'AAs': 'sequence', 'avg_mu-bg/
↪max': 'fitness'})
```

[ ]:
```
# remove star aas
df = df[~df['sequence'].apply(lambda x: '*' in x)]
```

1.2.2 0.2. Define wildtype sequence, get full sequences of variants

The wildtype in the study is not acutally the wildtype protein, they started with a variant. Define the true wt, apply mutations.

[ ]:
```
wt = ap.ProteinSequence(
    ␣
 ↪'MKGYFGPYGGQYVPEILMGALEELEAAYEGIMKDESFWKEFNDLLRDYAGRPTPLYFARRLSEKYGARVYLKREDLLHTGAHKINNAIGQ
    id='wt',
)
```

[ ]:
```
mutations = ['P18G', 'E29G', 'I68V', 'K95L', 'P139L', 'N166D', 'I183F',␣
 ↪'L212P', 'G227S', 'T291S']
wt = wt.mutate(mutations, one_indexed=False)
wt.id='Tm8D9'
```

[ ]:
```
msa = ap.ProteinSequences.from_fasta(os.path.join(RAW_DATA_DIR, 'EVMutation',␣
 ↪'TARGET_b0.1', 'align', 'TARGET_b0.1.a2m'))
```

[ ]:
```
wt.id = msa[0].id
```

[ ]:
```
msa_prc = MSAProcessing()
msa_sub = msa_prc.process(msa, focus_seq_id=wt.id)
```

[ ]:
```
selected_indices = np.random.choice(len(msa_sub), replace=False, size=1000,␣
 ↪p=msa_sub.weights/sum(msa_sub.weights))
```

[ ]:
```
selected_indices = [0] + list(selected_indices)
```

[ ]:
```
msa_sub = msa_sub[np.array(selected_indices)]
msa_sub.to_fasta('selected_msa.fasta')
```

[ ]:
```
msa_sub = ap.ProteinSequences.from_fasta('selected_msa.fasta')
wt = msa_sub[0]
```

[ ]:
```
library_positions = [182, 183, 226, 227]
mutation_strings_base = [
    f"{wt[pos]}{pos}" for pos in library_positions
]
def get_full_sequence(wt, aas):
    mutation_strings = [mutation_strings_base[i]+aa for i, aa in enumerate(aas)]
    return wt.mutate(mutation_strings, one_indexed=False)
df['full_sequence'] = df['sequence'].apply(lambda x: get_full_sequence(wt, x))
```

Shuffle the data and create X, y

[ ]:
```
df = df.sample(frac=1, random_state=0)
X = ap.ProteinSequences(df['full_sequence'].tolist())
y = df['fitness'].values
```

Plot the activity data

[ ]:
```
wt_activity = df[df['sequence'] == 'VFVS'].iloc[0]['fitness']
wt_activity
```

[ ]:
```
fig, ax = plt.subplots(figsize=(6, 6))
sns.kdeplot(y, ax=ax, bw_adjust=0.5)
ax.vlines(wt_activity, 0, ax.get_ylim()[1], color='red', label='WT activity')
ax.legend()
ax.set_xlabel('Fitness')
```

1.3 1. Data preprocessing

We need to: 1. Define splits for Kfold the better resemble actual training set sizes, eg. train on 130k variants, test on remaining 20k is not reasonable. Note that plenty of work on active learning suggest we can get accuracy with fewer samples by non random selection, but acquisition functions/active learning are currently native to AIDE. It is not hard to use AIDE models in an active learning loop, but we do not have boilerplate importable code for that yet so here we are doing more standard supervised model evaluation. 2. Define metrics that are relevant to the study, kendall's tau and top 10 recovery is a better representation of our experimental goal than spearman. 3. Define a wrapper function to run hyperparameter optimization and test set as training size increases, over random instatiation

1.3.1 1.1. Define splits

[ ]:
```
def modified_kfold_iterator(X, n_splits=5, shuffle=True, random_state=42,␣
↪training_size=96*100, test_size='all'):
    """KFold iterator that uses smaller training set and larger test set

    Params:
    -------
    X: array-like
        The data to split
    n_splits: int
        Number of splits
    shuffle: bool
        Whether to shuffle the data
    random_state: int
        Random state
    training_size: int
        Size of the training set, will be sampled from fold
    test_size: int or 'all'
        Size of the test set, if 'all' then all data not in training set will␣
↪be used
    """
    kf = KFold(n_splits=n_splits, shuffle=shuffle, random_state=random_state)
    np.random.seed(random_state)
    for test_index, train_index in kf.split(X):
```

```
        # sample to training size
        train_index = np.random.choice(train_index, training_size,␣
↪replace=False)
        if test_size != 'all':
            test_index = np.random.choice(test_index, test_size, replace=False)
        yield train_index, test_index
```

1.3.2 1.2. Define metrics

[ ]:
```
def top10_in_plate(y_true, y_pred):
    """Calculate top 10 recovery in a single plate, eg. how many of the top 10␣
↪true values are in the top 96 predicted values"""
    # now get top 10 recovery in a single plate
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    top_10_idx = list(np.argsort(y_true)[-10:])
    top_100_predicted_idx = list(np.argsort(y_pred)[-96:])
    top_10_recovery = sum([1 for idx in top_10_idx if idx in␣
↪top_100_predicted_idx]) / 10
    return top_10_recovery
```

[ ]:
```
scoring = {
    'kendall_tau': make_scorer(lambda y_true, y_pred: kendalltau(y_true,␣
↪y_pred)[0]),
    'top10_in_plate': make_scorer(top10_in_plate),
    'spearman': make_scorer(lambda y_true, y_pred: spearmanr(y_true,␣
↪y_pred)[0]),
}
```

[ ]:
```
def cv_1(model, X, y, training_size=96*10, random_state=42):
    """Cross validate a model with a single training size.
    """
    return cross_validate(
        model,
        X,
        y,
        cv=modified_kfold_iterator(X, training_size=training_size,␣
↪random_state=random_state),
        scoring=scoring,
    )

def cv_increasing_data_size(model, X, y, training_sizes=[24*2, 24*3, 24*4,␣
↪24*5, 24*6, 24*7, 24*8, 24*9, 24*10, 24*12, 24*16, 24*20, 24*24, 24*100],␣
↪n_repeats=20):
    """Cross validate a model with increasing training sizes, over a number of␣
↪repeats."""
```

```
    results = []
    for repeat in range(n_repeats):
        for training_size in training_sizes:
            results.append(cv_1(model, X, y, training_size=training_size,␣
 ↪random_state=repeat))
            results[-1]['training_size'] = training_size
    return results

def hyperopt_and_scoring(model, X, y, param_distributions,␣
 ↪ho_training_size=96*10, eval_training_sizes=[24*2, 24*3, 24*4, 24*5, 24*6,␣
 ↪24*7, 24*8, 24*9, 24*10, 24*12, 24*16, 24*20, 24*24, 24*100], n_repeats=20):
    """Hyperparameter optimization and scoring over increasing training sizes␣
 ↪in one call.

    Params
    ------
    model: sklearn estimator
        The model to optimize and score
    X: array-like
    y: array-like
    param_distributions: dict
        The hyperparameter distributions to search over, to be fed to␣
 ↪RandomizedSearchCV
    ho_training_size: int
        The size of the training set for hyperparameter optimization
    eval_training_sizes: list of ints
        The training sizes to evaluate the model on
    n_repeats: int
        Number of repeats to average over
    """
    searcher = RandomizedSearchCV(
        model,
        param_distributions=param_distributions,
        n_iter=100,
        n_jobs=1,
        cv=modified_kfold_iterator(X, training_size=ho_training_size),
        scoring=scoring,
        refit='kendall_tau',
    )
    searcher.fit(X, y)
    best_params = searcher.best_params_
    scores = cv_increasing_data_size(model.set_params(**best_params), X, y,␣
 ↪training_sizes=eval_training_sizes, n_repeats=n_repeats)
    return scores
```

1.4 2. Self supervised predictors

1.4.1 2.1 ESM2 with wildtype marginal

[ ]:
```
esm2liklihood = ap.ESM2LikelihoodWrapper(
    wt=wt,
    marginal_method='wildtype_marginal',
    device='mps',
    metadata_folder=os.path.join('.', 'data', 'epistatic', 'esm2likelihood'),
    model_checkpoint='esm2_t33_650M_UR50D',
    use_cache=True,
)
esm2liklihood.fit([])
```

[ ]:
```
esm_none_none_scores = cv_1(esm2liklihood, X, y)
```

[ ]:
```
joblib.dump(esm_none_none_scores, 'esm_none_none_scores.joblib')
```

1.4.2 2.2 EVC

[ ]:
```
evc = ap.EVMutationWrapper(
    wt=wt,
    metadata_folder=os.path.join('.', 'data', 'epistatic', 'evmutation'),
)
evc.fit(msa_sub)
```

[ ]:
```
evc_none_none_scores = cv_1(evc, X, y)
```

[ ]:
```
joblib.dump(evc_none_none_scores, 'evc_none_none_scores.joblib')
```

1.4.3 2.3 MSATransformer

[ ]:
```
msa_likelihood = ap.MSATransformerLikelihoodWrapper(
    wt=wt,
    metadata_folder=os.path.join('.', 'data', 'epistatic', 'msalikelihood'),
    device='mps',
    marginal_method='wildtype_marginal',
)
msa_likelihood.fit(msa_sub)
```

[ ]:
```
msa_none_none_scores = cv_1(msa_likelihood, X, y)
```

[ ]:
```
joblib.dump(msa_none_none_scores, 'msa_none_none_scores.joblib')
```

1.5 3. Supervised learning

1.5.1 3.0 downstream pipelines

We can specifiy embedders with AIDE into a scikitlearn pipeline, but to avoid recomputation of embeddings we will precompute them and pass them into specified pipelines.

[ ]:
```
linear_nopca = Pipeline([
    ('scaler', StandardScaler()),
    ('model', TransformedTargetRegressor(
        regressor=Ridge(
            alpha=1.0,
        )))])
linear_pca = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.9)),
    ('model', TransformedTargetRegressor(
        regressor=Ridge(
            alpha=1.0,
        )))])
linear_param_space = {
    'model__regressor__alpha': loguniform(1e-3, 1e3),
}
```

[ ]:
```
mlp_nopca = Pipeline([
    ('scaler', StandardScaler()),
    ('model', TransformedTargetRegressor(
        regressor=MLPRegressor(
            hidden_layer_sizes=(20, 20),
            max_iter=1000,
            early_stopping=True
        )))])
mlp_pca = Pipeline([
    ('scaler', StandardScaler()),
    ('pca', PCA(n_components=0.9)),
    ('model', TransformedTargetRegressor(
        regressor=MLPRegressor(
            hidden_layer_sizes=(20, 20),
            max_iter=1000,
            early_stopping=True
        )))])
mlp_param_space = {
    'model__regressor__hidden_layer_sizes': [(20,), (20, 20), (20, 20, 20),␣
 ↪(20, 20, 20, 20)],
    'model__regressor__alpha': loguniform(1e-6, 1e-1),
    'model__regressor__activation': ['relu', 'tanh'],
}
```

1.5.2 3.1 OHE linear

[ ]:
```
ohe = ap.OneHotProteinEmbedding(
    positions=library_positions,
    metadata_folder=os.path.join('.', 'data', 'epistatic', 'onehot'))
X_ohe = ohe.fit_transform(X)
```

[ ]:
```
none_ohe_linear_scores = hyperopt_and_scoring(linear_nopca, X_ohe, y,␣
 ↪linear_param_space)
joblib.dump(none_ohe_linear_scores, 'none_ohe_linear_scores.joblib')
```

1.5.3 3.2 OHE MLP

[ ]:
```
none_ohe_mlp_scores = hyperopt_and_scoring(mlp_nopca, X_ohe, y, mlp_param_space)
joblib.dump(none_ohe_mlp_scores, 'none_ohe_mlp_scores.joblib')
```

1.5.4 3.3 ESM2 embeddings, full sequence, linear model

Include scaling and PCA to reduce dims.

[ ]:
```
esm2_embedder = ap.ESM2Embedding(
    pool='mean',
    metadata_folder=os.path.join('.', 'data', 'epistatic', 'esm2emb_full'),
    device='mps',
    model_checkpoint='esm2_t12_35M_UR50D',
    use_cache=True,
    batch_size=180
)
esm2_embedder.fit([])
X_esm = esm2_embedder.transform(X)
```

[ ]:
```
none_esmfull_linear_scores = hyperopt_and_scoring(linear_pca, X_esm, y,␣
 ↪linear_param_space)
joblib.dump(none_esmfull_linear_scores, 'none_esmfull_linear_scores.joblib')
```

1.5.5 3.4 ESM2 embeddings, full sequence, MLP

[ ]:
```
none_esmfull_mlp_scores = hyperopt_and_scoring(mlp_pca, X_esm, y,␣
 ↪mlp_param_space)
joblib.dump(none_esmfull_mlp_scores, 'none_esmfull_mlp_scores.joblib')
```

1.5.6 3.5 ESM2 embeddings, only changing residues, linear

[ ]:
```
esm2_embedder_only4 = ap.ESM2Embedding(
    pool='mean',
    metadata_folder=os.path.join('.', 'data', 'epistatic',␣
 ↪'esm2emb_design_space'),
    device='mps',
```

```
    model_checkpoint='esm2_t12_35M_UR50D',
    use_cache=True,
    batch_size=180,
    positions=library_positions
)
esm2_embedder_only4.fit([])
X_esm_only4 = esm2_embedder_only4.transform(X)
```

[ ]:
```
none_esm4sites_linear_scores = hyperopt_and_scoring(linear_pca, X_esm_only4, y,␣
↪linear_param_space)
joblib.dump(none_esm4sites_linear_scores, 'none_esm4sites_linear_scores.joblib')
```

1.5.7 3.6 ESM2 embeddings, only changing residues, MLP

[ ]:
```
none_esm4sites_mlp_scores = hyperopt_and_scoring(mlp_pca, X_esm_only4, y,␣
↪mlp_param_space)
joblib.dump(none_esm4sites_mlp_scores, 'none_esm4sites_mlp_scores.joblib')
```

1.6 4. Combined models

1.6.1 4.1 EVC plus one hot linear

[ ]:
```
X_evc = evc.transform(X)
```

[ ]:
```
X_ = np.concatenate([X_ohe, X_evc], axis=1)
```

[ ]:
```
evc_ohe_linear_scores = hyperopt_and_scoring(linear_nopca, X_, y,␣
↪linear_param_space)
joblib.dump(evc_ohe_linear_scores, 'evc_ohe_linear_scores.joblib')
```

1.6.2 4.2 EVC plus one hot MLP

[ ]:
```
evc_ohe_mlp_scores = hyperopt_and_scoring(mlp_nopca, X_, y, mlp_param_space)
joblib.dump(evc_ohe_mlp_scores, 'evc_ohe_mlp_scores.joblib')
```

1.6.3 4.3 EVC plus ESM2 linear

[ ]:
```
X_ = np.concatenate([X_esm, X_evc], axis=1)
```

[ ]:
```
evc_esmfull_linear_scores = hyperopt_and_scoring(linear_pca, X_, y,␣
↪linear_param_space)
joblib.dump(evc_esmfull_linear_scores, 'evc_esmfull_linear_scores.joblib')
```

1.6.4 4.4 EVC plus ESM2 MLP

[ ]:
```
evc_esmfull_mlp_scores = hyperopt_and_scoring(mlp_pca, X_, y, mlp_param_space)
joblib.dump(evc_esmfull_mlp_scores, 'evc_esmfull_mlp_scores.joblib')
```

1.6.5 4.5 EVC plus ESM2 only 4 linear

[ ]:
```
X_ = np.concatenate([X_esm_only4, X_evc], axis=1)
```

[ ]:
```
evc_esm4sites_linear_scores = hyperopt_and_scoring(linear_pca, X_, y,␣
↪linear_param_space)
joblib.dump(evc_esm4sites_linear_scores, 'evc_esm4sites_linear_scores.joblib')
```

1.6.6 4.6 EVC plus ESM2 only 4 MLP

[ ]:
```
evc_esm4sites_mlp_scores = hyperopt_and_scoring(mlp_pca, X_, y, mlp_param_space)
joblib.dump(evc_esm4sites_mlp_scores, 'evc_esm4sites_mlp_scores.joblib')
```

1.7 5. Compare and visualize results

[ ]:
```
zs_scores = {
    'evc_none_none': joblib.load('evc_none_none_scores.joblib'),
    'esm_none_none': joblib.load('esm_none_none_scores.joblib'),
    'msa_none_none': joblib.load('msa_none_none_scores.joblib'),
}
supervised_scores = {
    'none_ohe_linear': joblib.load('none_ohe_linear_scores.joblib'),
    'none_ohe_mlp': joblib.load('none_ohe_mlp_scores.joblib'),
    'none_esmfull_linear': joblib.load('none_esmfull_linear_scores.joblib'),
    'none_esmfull_mlp': joblib.load('none_esmfull_mlp_scores.joblib'),
    'none_esm4sites_linear': joblib.load('none_esm4sites_linear_scores.joblib'),
    'none_esm4sites_mlp': joblib.load('none_esm4sites_mlp_scores.joblib'),
    'evc_ohe_linear': joblib.load('evc_ohe_linear_scores.joblib'),
    'evc_ohe_mlp': joblib.load('evc_ohe_mlp_scores.joblib'),
    'evc_esmfull_linear': joblib.load('evc_esmfull_linear_scores.joblib'),
    'evc_esmfull_mlp': joblib.load('evc_esmfull_mlp_scores.joblib'),
    'evc_esm4sites_linear': joblib.load('evc_esm4sites_linear_scores.joblib'),
    'evc_esm4sites_mlp': joblib.load('evc_esm4sites_mlp_scores.joblib'),
}
```

[ ]:
```
data = []
for model, score_list in supervised_scores.items():
    for score_dict in score_list:
        training_size = score_dict['training_size']
        for metric in ['kendall_tau', 'spearman', 'top10_in_plate']:
            for point in score_dict[f'test_{metric}']:
                data.append({
```

```
                    'Model': model,
                    'Metric': metric,
                    'Score': point,
                    'Training Size': training_size
                })

supervised_df = pd.DataFrame(data)
supervised_df[['zs', 'embedding', 'topmodel']] = supervised_df['Model'].str.
 ↪split('_', expand=True)
data = []
for model, scores in zs_scores.items():
    for metric in ['kendall_tau', 'spearman', 'top10_in_plate']:
        for point in scores[f'test_{metric}']:
            data.append({
                'Model': model,
                'Metric': metric,
                'Score': point,
                'Training Size': 0.0
            })
zs_df = pd.DataFrame(data)
zs_df[['zs', 'embedding', 'topmodel']] = zs_df['Model'].str.split('_',␣
 ↪expand=True)
```

[ ]:
```
# determined by embedding type
color_map = {
    'none': 'grey',
    'ohe': 'tab:blue',
    'esmfull': 'tab:orange',
    'esm4sites': 'tab:green',
}
# determined by ZS method
linestyle_map = {
    'none': ':',
    'evc': '-',
    'esm': '--',
    'msa': '-.',
}
# determined by top model
marker_map = {
    'none': 'o',
    'linear': 's',
    'mlp': 'd',
}
```

[ ]:
```
def get_name_logic(string):
    split = string.split('_')
```

```
    zs_input = split[0]
    embedding = split[1]
    topmodel = split[2]
    if embedding == 'none' or topmodel == 'none':
        type_ = 'ZS'
    elif zs_input != 'none':
        type_ = 'Aug.'
    else:
        type_ = 'Sup.'

    output = ''
    if zs_input != 'none':
        output += zs_input + ''
    if embedding != 'none':
        if len(output) > 0:
            output += '+'
        output += embedding + ''
    if topmodel != 'none':
        output += '->'
        output += topmodel

    output += ' (' + type_ + ')'
    return output
```

[ ]:
```
supervised_df['name'] = supervised_df.apply(lambda x:␣
 ↪get_name_logic(x['Model']), axis=1)
zs_df['name'] = zs_df.apply(lambda x: get_name_logic(x['Model']), axis=1)
```

[ ]:
```
# create plot of ZS vs supervised vs augmented linear, like original paper
metric = 'kendall_tau'

fig, ax = plt.subplots(figsize=(6, 6))

for model, df in supervised_df.groupby('Model'):
    df_ = df[df['Metric'] == metric]
    color, linestyle, marker = color_map[model.split('_')[1]],␣
 ↪linestyle_map[model.split('_')[0]], marker_map[model.split('_')[2]]
    if marker == 'd':
        continue
    name = df_['name'].iloc[0]

    sns.lineplot(x='Training Size', y='Score', data=df_, ax=ax,
                 label=name, color=color, linestyle=linestyle, marker=marker)

for model, df in zs_df.groupby('Model'):
    df_ = df[df['Metric'] == metric]
```

```
    color, linestyle, marker = color_map[model.split('_')[1]],␣
 ↪linestyle_map[model.split('_')[0]], marker_map[model.split('_')[2]]
    name = df_['name'].iloc[0]
    xmin = ax.get_xlim()[0]
    xmax = ax.get_xlim()[1]
    mean = df_.mean()['Score']
    ax.hlines(mean, xmin, xmax, color=color, linestyle=linestyle, label=name)

ax.set_xlabel('Training Size')
ax.set_ylabel("Kendall's Tau")
plt.legend(loc='upper right', bbox_to_anchor=(1.9, .8))
plt.savefig('zs_vs_supervised_vs_augmented.png', bbox_inches='tight', dpi=300)
```

[ ]:
```
# now compare linear to nonlinear models
metric = 'kendall_tau'

fig, ax = plt.subplots(figsize=(6, 6))

for model, df in supervised_df.groupby('Model'):
    df_ = df[df['Metric'] == metric]
    color, linestyle, marker = color_map[model.split('_')[1]],␣
 ↪linestyle_map[model.split('_')[0]], marker_map[model.split('_')[2]]
    if linestyle != ':':
        continue
    name = df_['name'].iloc[0]
    sns.lineplot(x='Training Size', y='Score', data=df_, ax=ax,
                 label=name, color=color, linestyle=linestyle, marker=marker)

ax.set_xlabel('Training Size')
ax.set_ylabel("Kendall's Tau")
plt.legend(loc='upper right', bbox_to_anchor=(1.8, .8))
plt.savefig('nonlinear_vs_linear_kendall.png', bbox_inches='tight', dpi=300)
```

[ ]:
```
# now compare linear to nonlinear models
metric = 'top10_in_plate'

fig, ax = plt.subplots(figsize=(6, 6))

for model, df in supervised_df.groupby('Model'):
    df_ = df[df['Metric'] == metric]
    color, linestyle, marker = color_map[model.split('_')[1]],␣
 ↪linestyle_map[model.split('_')[0]], marker_map[model.split('_')[2]]
    if model.split('_')[0] != 'none':
        continue

    name = df_['name'].iloc[0]
    sns.lineplot(x='Training Size', y='Score', data=df_, ax=ax,
```

```
                    label=name, color=color, linestyle=linestyle, marker=marker)

ax.set_xlabel('Training Size')
ax.set_ylabel("Top10 Recovery, 1 96w-plate")
plt.legend()
plt.savefig('nonlinear_vs_linear_top10.png', bbox_inches='tight', dpi=300)
```

[ ]:

[ ]:

[ ]: