Portfolio Optimization of Factor-Based Risk Model using Machine Learning

December 23, 2024

0.0.1 Problem 0.

All of the below pertain to the estimation universe as defined above. Modify the daily data frames, removing all non-estimation-universe rows, before continuing.

[1]:
```
import pandas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import pickle
from statsmodels.formula.api import ols
from scipy.stats import gaussian_kde
import scipy
import scipy.sparse
from scipy.linalg import pinv
import patsy
from statistics import median
import bz2
import math
```

[2]:
```
model_dir = '/Users/kechengshi/Documents/Python/MATH5430/FACTOR_MODEL/'

def sort_cols(test):
    return(test.reindex(sorted(test.columns), axis=1))

frames = {}
for year in list(range(2003,2011)):
    fil = model_dir + "pandas-frames." + str(year) + ".pickle.bz2"
    frames.update(pd.read_pickle(fil))

for x in frames:
    frames[x] = sort_cols(frames[x])

covariance = {}

for year in list(range(2003,2011)):
    fil = model_dir + "covariance." + str(year) + ".pickle.bz2"
    covariance.update(pd.read_pickle(fil))
```

[3]:
```python
industry_factors = ['AERODEF', 'AIRLINES', 'ALUMSTEL', 'APPAREL', 'AUTO',
'BANKS','BEVTOB', 'BIOLIFE', 'BLDGPROD','CHEM', 'CNSTENG',
'CNSTMACH', 'CNSTMATL', 'COMMEQP', 'COMPELEC',
'COMSVCS', 'CONGLOM', 'CONTAINR', 'DISTRIB',
'DIVFIN', 'ELECEQP', 'ELECUTIL', 'FOODPROD', 'FOODRET', 'GASUTIL',
'HLTHEQP', 'HLTHSVCS', 'HOMEBLDG', 'HOUSEDUR','INDMACH', 'INSURNCE','INTERNET',
'LEISPROD', 'LEISSVCS', 'LIFEINS', 'MEDIA', 'MGDHLTH','MULTUTIL',
'OILGSCON', 'OILGSDRL', 'OILGSEQP', 'OILGSEXP',
'PAPER', 'PHARMA', 'PRECMTLS','PSNLPROD','REALEST',
'RESTAUR', 'ROADRAIL','SEMICOND', 'SEMIEQP','SOFTWARE',
'SPLTYRET', 'SPTYCHEM', 'SPTYSTOR', 'TELECOM', 'TRADECO', 'TRANSPRT','WIRELESS']
style_factors = ['BETA','SIZE','MOMENTUM','VALUE','LEVERAGE','LIQUIDTY']
```

[4]:
```python
def wins(x,a,b):
    return(np.where(x <= a,a, np.where(x >= b, b, x)))
def clean_nas(df):
    numeric_columns = df.select_dtypes(include=[np.number]).columns.tolist()

    for numeric_column in numeric_columns:
        df[numeric_column] = np.nan_to_num(df[numeric_column])

    return df
```

[5]:
```python
def get_estu(df):
    """Estimation universe definition"""
    estu = df.loc[df.IssuerMarketCap > 1e9].copy(deep=True)
    return estu

def colnames(X):
    """ return names of columns, for DataFrame or DesignMatrix """
    if(type(X) == patsy.design_info.DesignMatrix):
        return(X.design_info.column_names)
    if(type(X) == pandas.core.frame.DataFrame):
        return(X.columns.tolist())
    return(None)

def diagonal_factor_cov(date, X):
    """Factor covariance matrix, ignoring off-diagonal for simplicity"""
    cv = covariance[date]
    k = np.shape(X)[1]
    Fm = np.zeros([k,k])
    for j in range(0,k):
        fac = colnames(X)[j]
        Fm[j,j] = (0.01**2) * cv.loc[(cv.Factor1==fac) & (cv.
↪Factor2==fac),"VarCovar"].iloc[0]
    return(Fm)
```

```python
def risk_exposures(estu):
    """Exposure matrix for risk factors, usually called X in class"""
    L = ["0"]
    L.extend(style_factors)
    L.extend(industry_factors)
    my_formula = " + ".join(L)
    return patsy.dmatrix(my_formula, data = estu)
```

[6]:
```python
# Filter each daily frame for estimation universe
for date, df in frames.items():
    frames[date] = get_estu(df)
```

0.0.2 Problem 1.

Residual returns Within each daily data frame, let **Y** denote the residuals of the variable **Ret**, with respect to the risk model. In other words, define

$$Y := \text{Ret} - XX^{+}\text{Ret}$$

where X^{+} denotes the pseudoinverse, and X is constructed as above (i.e., using the risk_exposures function). Augment the data frames you have been given, by adding a new column, Y, to each frame. Be sure to winsorize the *Ret* column prior to computing Y as above. You do not have to save the augmented data, unless you want to. In other words, the modification that adds column Y can be done in-memory.

[7]:
```python
for date, df in frames.items():
    # Winsorize Ret
    df["Ret"] = wins(df["Ret"], -0.25, 0.25)

    # Compute risk exposure matrix
    X = risk_exposures(df)

    # Compute residual returns
    df["Y"] = df["Ret"] - np.dot(np.dot(X, pinv(X)), df["Ret"])
```

0.0.3 Problem 2.

Model selection Split your data into a training/validation set D_{train}, and an ultimate test set (vault), D_{test}. Do not split within a single day; rather, some dates end up in D_{train} and the rest in D_{test}. This will be the basis of your cross-validation study later on.

It will be helpful to join together vertically the frames in the training/validation set D_{train} into a single frame called a panel. For the avoidance of doubt, the panel will have the same columns as any one of the daily frames individually, and the panel will have a large number of rows (the sum of all the rows of all the frames in D_{train}).

Consider list of candidate alpha factors given above. Find a model of the form

$$Y = f(\text{candidate alphas}) + \epsilon$$

where Y is the residual return from above. Determine the function $f()$ using cross-validation to optimize any tunable hyper-parameters. First, to get started, assume f is linear and use lasso or elastic net cross-validation tools (e.g. from sklearn). Then, get creative and try at least one non-linear functional form for f, again using cross-validation to optimize any tunable hyper-parameters.

Weighted Least Squares

[8]:
```
# Combine all daily frames into a single panel
panel = pd.concat(frames.values(), keys=frames.keys(), names=["Date", "Index"])
del frames,df, X
```

[9]:
```
# Split by dates: 80% training, 20% testing
dates = panel.index.get_level_values("Date").unique()
train_dates = dates[:int(0.8 * len(dates))]
test_dates = dates[int(0.8 * len(dates)):]

# Create train and test sets
train_panel = panel.loc[train_dates]
test_panel = panel.loc[test_dates]

# Candidate alpha factors
candidate_alphas = ['STREVRSL', 'LTREVRSL', 'INDMOM', 'EARNQLTY',
                    'EARNYILD', 'MGMTQLTY', 'PROFIT', 'SEASON', 'SENTMT']

# Extract data for training
Y_train = train_panel["Y"].to_numpy()  # Residual returns
X_train = np.asarray(train_panel[candidate_alphas])  # Alpha factors as a NumPy␣
 ↪array=
D_train = np.asarray((train_panel["SpecRisk"] / (100 * math.sqrt(252))) ** 2) ␣
 ↪# Specific risk diagonal

# Extract data for testing
Y_test = test_panel["Y"].to_numpy()  # Residual returns
X_test = np.asarray(test_panel[candidate_alphas])  # Alpha factors as a NumPy␣
 ↪array
D_test = np.asarray((test_panel["SpecRisk"] / (100 * math.sqrt(252))) ** 2)  #␣
 ↪Specific risk diagonal
```

[10]:
```
del panel
```

[11]:
```
def regularized_wls_train(X, Y, D_diag, kappa=1e-5):
    """
    Train a regularized WLS model with memory-efficient diagonal representation␣
 ↪of D.
```

```
    Parameters:
    X : ndarray
        Feature matrix (n x p).
    Y : ndarray
        Target variable (n-dimensional vector).
    D_diag : ndarray
        Diagonal elements of the specific risk matrix (1D array).
    kappa : float
        Regularization parameter.

    Returns:
    coef : ndarray
        Estimated coefficients (p-dimensional vector).
    """
    # Compute D^{-1} as element-wise inverse of D_diag
    D_inv_diag = 1 / D_diag  # Inverse of diagonal elements

    # Efficient computation of X^T D^{-1}
    X_t_D_inv = (D_inv_diag[:, None] * X).T  # Element-wise multiplication

    # Regularization term
    regularization = kappa * np.identity(X.shape[1])

    # Compute coefficients
    coef = np.linalg.inv(X_t_D_inv @ X + regularization) @ (X_t_D_inv @ Y)
    return coef
```

[12]:
```
def predict_wls(X, coef):
    """
    Make predictions using the WLS model.

    Parameters:
    X : ndarray
        Feature matrix (n x p).
    coef : ndarray
        Coefficients (p-dimensional vector).

    Returns:
    predictions : ndarray
        Predicted values (n-dimensional vector).
    """
    return X @ coef

from sklearn.metrics import mean_squared_error

# Regularization parameter
kappa = 1e-5
```

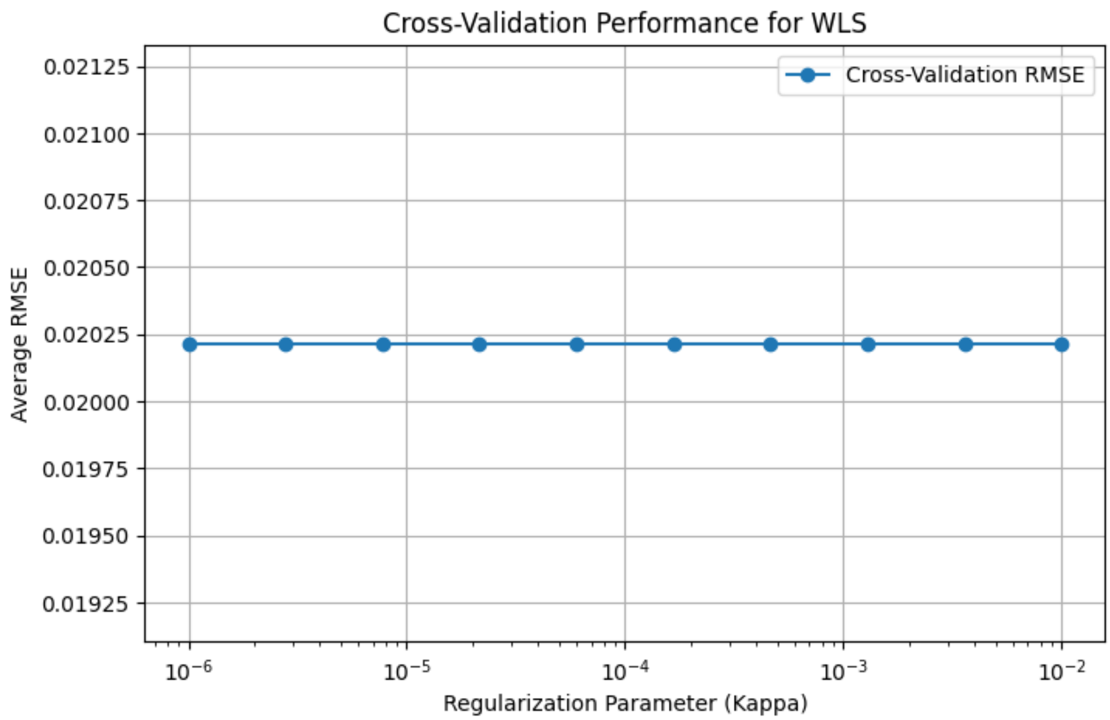
```python
# Train the model
coefficients = regularized_wls_train(X_train, Y_train, D_train, kappa)

# Make predictions
Y_train_pred = predict_wls(X_train, coefficients)
Y_test_pred = predict_wls(X_test, coefficients)

# Evaluate performance
train_rmse = np.sqrt(mean_squared_error(Y_train, Y_train_pred))
test_rmse = np.sqrt(mean_squared_error(Y_test, Y_test_pred))

print(f"Train RMSE: {train_rmse}")
print(f"Test RMSE: {test_rmse}")
```

```
Train RMSE: 0.020820101356573598
Test RMSE: 0.01930647219558436
```

[13]:
```python
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt

def cross_validate_wls_with_plot(X, Y, D_diag, kappa_values, cv=5):
    """
    Cross-validate to find the best kappa for regularized WLS with␣
↪memory-efficient D representation
    and plot performance.

    Parameters:
    X : ndarray
        Feature matrix (n x p).
    Y : ndarray
        Target variable (n-dimensional vector).
    D_diag : ndarray
        Diagonal elements of the specific risk matrix (1D array).
    kappa_values : list
        List of kappa values to test.
    cv : int
        Number of cross-validation folds.

    Returns:
    best_kappa : float
        Best regularization parameter.
    """
    kf = KFold(n_splits=cv)
    best_rmse = float("inf")
```

```
    best_kappa = None
    kappa_rmse = []  # To store average RMSE for each kappa

    for kappa in kappa_values:
        fold_rmses = []
        for train_index, val_index in kf.split(X):
            X_train, X_val = X[train_index], X[val_index]
            Y_train, Y_val = Y[train_index], Y[val_index]
            D_diag_train = D_diag[train_index]  # Use 1D representation

            # Train WLS model
            coef = regularized_wls_train(X_train, Y_train, D_diag_train, kappa)
            Y_val_pred = predict_wls(X_val, coef)

            # Evaluate RMSE
            fold_rmse = np.sqrt(mean_squared_error(Y_val, Y_val_pred))
            fold_rmses.append(fold_rmse)

        avg_rmse = np.mean(fold_rmses)
        kappa_rmse.append(avg_rmse)  # Track performance

        if avg_rmse < best_rmse:
            best_rmse = avg_rmse
            best_kappa = kappa

    # Plot RMSE vs Kappa
    plt.figure(figsize=(8, 5))
    plt.plot(kappa_values, kappa_rmse, marker='o', linestyle='-',␣
 ↪label="Cross-Validation RMSE")
    plt.xscale('log')  # Logarithmic scale for kappa
    plt.xlabel("Regularization Parameter (Kappa)")
    plt.ylabel("Average RMSE")
    plt.title("Cross-Validation Performance for WLS")
    plt.legend()
    plt.grid()
    plt.show()

    return best_kappa
```

[14]:
```
# Define kappa values to test
kappa_values = np.logspace(-6, -2, 10)

# Perform cross-validation with performance tracking
best_kappa = cross_validate_wls_with_plot(X_train, Y_train, D_train,␣
 ↪kappa_values)
print(f"Best Kappa: {best_kappa}")
```




```
Best Kappa: 1e-06
```

Neural Network

```
[15]: from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Define the parameter grid for the neural network
param_grid = {
    "hidden_layer_sizes": [(50,), (100,), (100, 50)],  # Different architectures
    "alpha": [0.0001, 0.001],  # Regularization strength
    "learning_rate_init": [0.001],  # Learning rate
}

# Initialize MLP Regressor with reduced max_iter
mlp = MLPRegressor(max_iter=200, random_state=42)  # Set random state for␣
 ↪reproducibility

# Use GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(
```

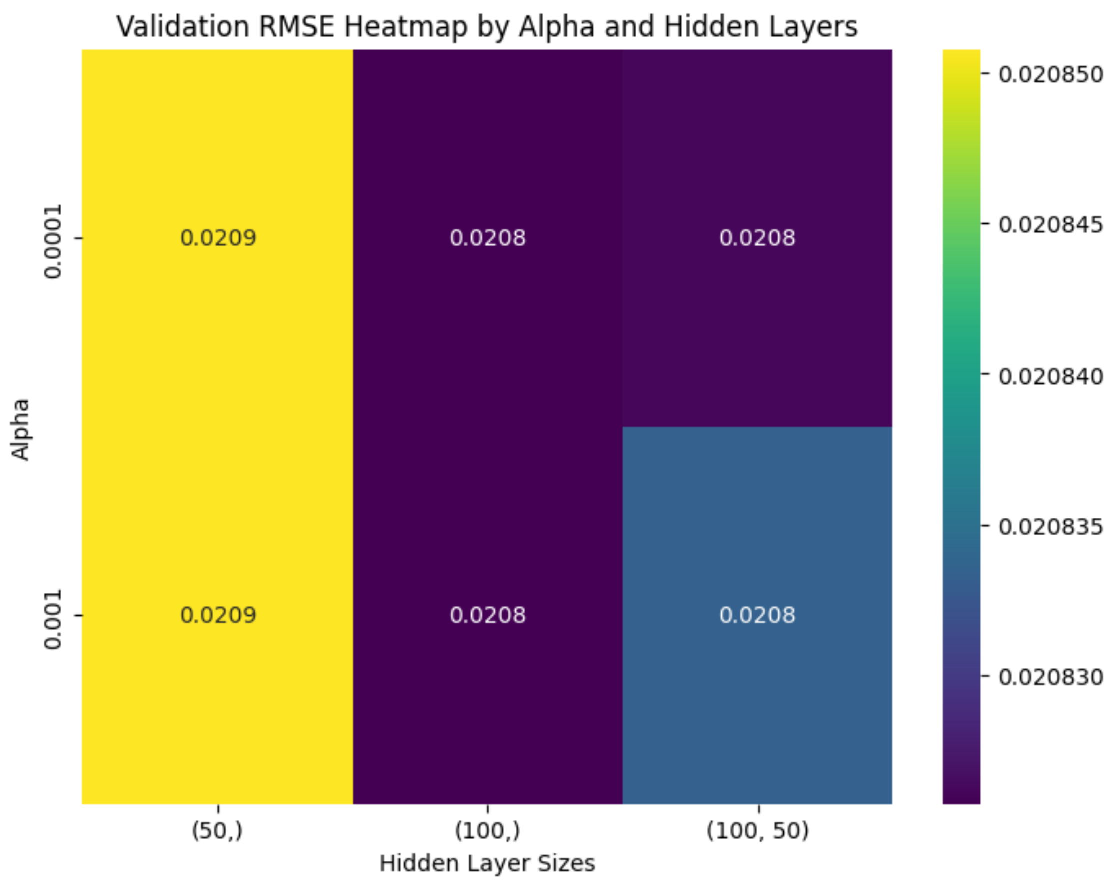
```python
    estimator=mlp,
    param_grid=param_grid,
    scoring="neg_mean_squared_error",
    cv=3,  # 3-fold cross-validation
    n_jobs=-1,  # Use all available CPU cores
    verbose=1,  # Detailed output during the grid search
)

# Fit the model using training data
grid_search.fit(X_train, Y_train)

# Get best parameters and the corresponding model
best_params = grid_search.best_params_
nn_model = grid_search.best_estimator_

# Print best parameters and validation RMSE
print(f"Best Parameters: {best_params}")
val_rmse = np.sqrt(-grid_search.best_score_)
print(f"Validation RMSE: {val_rmse}")

# Evaluate the model on the test set
Y_test_pred = nn_model.predict(X_test)
test_rmse = np.sqrt(mean_squared_error(Y_test, Y_test_pred))
print(f"Test RMSE: {test_rmse}")

# Extract grid search results for visualization
results = grid_search.cv_results_
hidden_layers = results["param_hidden_layer_sizes"].data
alphas = results["param_alpha"].data
mean_rmse = np.sqrt(-results["mean_test_score"])  # Convert negative MSE to RMSE

# Create DataFrame for heatmap visualization
df_results = pd.DataFrame({
    "Hidden Layers": hidden_layers,
    "Alpha": alphas,
    "Validation RMSE": mean_rmse,
})

# Pivot the DataFrame to plot a heatmap
pivot_table = df_results.pivot_table(index="Alpha", columns="Hidden Layers",␣
 ↪values="Validation RMSE")

# Plot the heatmap for performance
plt.figure(figsize=(8, 6))
sns.heatmap(pivot_table, annot=True, fmt=".4f", cmap="viridis")
plt.title("Validation RMSE Heatmap by Alpha and Hidden Layers")
plt.xlabel("Hidden Layer Sizes")
```

```
plt.ylabel("Alpha")
plt.show()
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
```

```
/opt/anaconda3/lib/python3.12/site-
packages/sklearn/neural_network/_multilayer_perceptron.py:698: UserWarning:
Training interrupted by user.
  warnings.warn("Training interrupted by user.")
```

```
Best Parameters: {'alpha': 0.0001, 'hidden_layer_sizes': (100,),
'learning_rate_init': 0.001}
Validation RMSE: 0.02082573772448532
Test RMSE: 0.019317893755487342
```



0.0.4 Problem 3.

Efficient portfolio optimization Code up the efficient formula for portfolio optimization discussed in lecture, based on the Woodbury matrix inversion lemma.

We aim to compute optimal portfolio weights w using the formula:

$$w = \Sigma^{-1}\mu$$

where Σ is defined as:

$$\Sigma = D + XFX^\top$$

To avoid division by zero, replace zero or near-zero values in D and F with a small positive constant (1×10^{-10}):

$$D_{\text{diag}} = \max(D_{\text{diag}}, 1 \times 10^{-10}), \quad F_{\text{diag}} = \max(F_{\text{diag}}, 1 \times 10^{-10})$$

The inverses of the diagonal matrices D and F are calculated as:

$$D_{\text{diag}}^{-1} = \frac{1}{D_{\text{diag}}}, \quad F_{\text{diag}}^{-1} = \frac{1}{F_{\text{diag}}}$$

Using the diagonal inverse of D, compute:

$$D^{-1}X = D_{\text{diag}}^{-1} \cdot X$$

This is achieved using element-wise multiplication, avoiding the construction of full matrices.

The middle term in the Woodbury formula is:

$$\text{Middle Term} = \left(F^{-1} + X^\top D^{-1} X\right)^{-1}$$

Here, the F^{-1} matrix is diagonal, so we use:

$$F^{-1} = \text{diag}(F_{\text{diag}}^{-1})$$

Using the Woodbury matrix inversion lemma:

$$\Sigma^{-1}\mu = D_{\text{diag}}^{-1}\mu - D_{\text{diag}}^{-1}X \, \text{Middle Term} \, X^\top D_{\text{diag}}^{-1}\mu$$

The resulting portfolio weights w are given by:

$$w = \Sigma^{-1}\mu$$

This provides the optimal weights for the portfolio based on the given inputs.

[16]:
```
def compute_portfolio_weights(X, F_diag, D_diag, mu):
    """
    Compute optimal portfolio weights using the Woodbury matrix inversion lemma␣
↪with memory-efficient diagonal representations.
```

```python
    Parameters:
    X : ndarray
        Factor exposures matrix (n x p).
    F_diag : ndarray
        Diagonal elements of the factor covariance matrix (1D array, size p).
    D_diag : ndarray
        Diagonal elements of the specific risk covariance matrix (1D array,␣
↪size n).
    mu : ndarray
        Expected returns (n-dimensional vector).

    Returns:
    w : ndarray
        Optimal portfolio weights (n-dimensional vector).
    """
    # Replace zero or near-zero values with small positive values for stability
    D_diag = np.where(D_diag > 0, D_diag, 1e-10)
    F_diag = np.where(F_diag > 0, F_diag, 1e-10)

    # Compute D^{-1} and F^{-1}
    D_inv_diag = 1 / D_diag
    F_inv_diag = 1 / F_diag

    # Compute D^{-1} X efficiently
    D_inv_X = D_inv_diag[:, None] * X  # Element-wise multiplication

    # Compute the middle term using pseudo-inverse for stability
    middle_term = np.linalg.pinv(np.diag(F_inv_diag) + X.T @ D_inv_X)

    # Compute Sigma^{-1} using the Woodbury formula
    Sigma_inv_mu = D_inv_diag * mu - (D_inv_diag * (X @ middle_term @ (X.T @␣
↪(D_inv_diag * mu))))

    return Sigma_inv_mu
```

0.0.5 Problem 4.

Putting it all together Using the helpful code example above, and using the output of the function $ f $ as your final alpha factor, construct a backtest of a portfolio optimization strategy. In other words, compute the optimal portfolio each day, and dot product it with Ret to get the pre-tcost 1-day profit for each day. Use the previous problem to speed things up. Create time-series plots of the long market value, short market value, and cumulative profit of this portfolio sequence. Also plot the daily risk, in dollars, of your portfolios and the percent of the risk that is idiosyncratic.

Weighted Least Squares

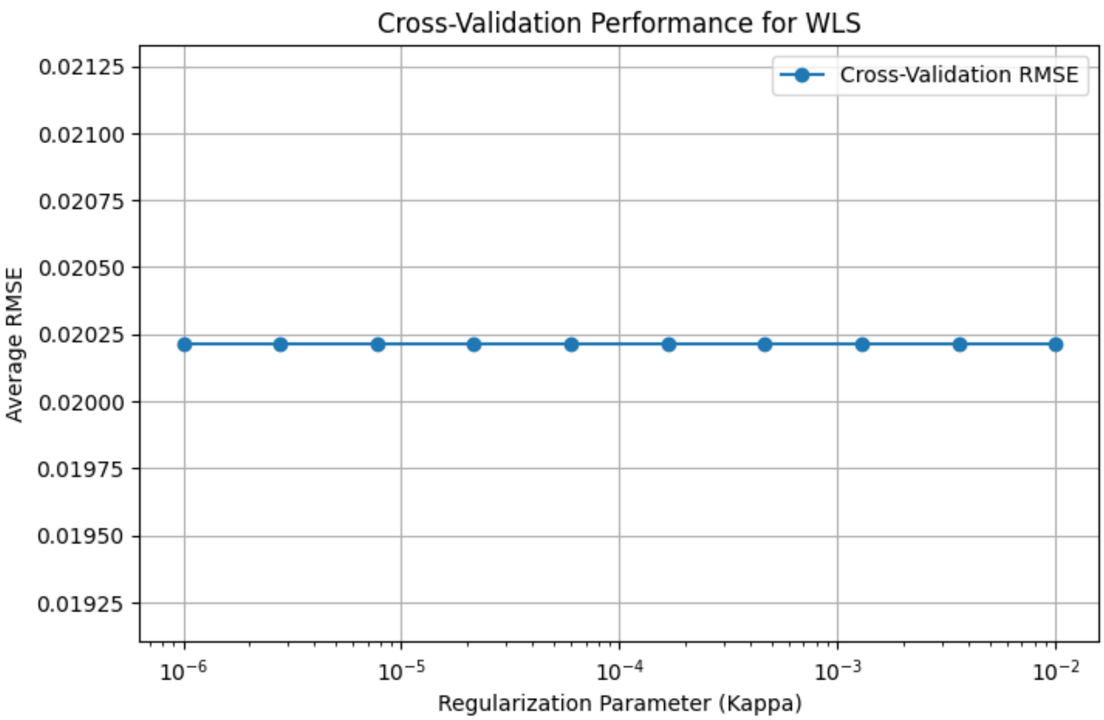
```
[17]: def backtest_with_train_test(
          train_panel, test_panel, Y_train, X_train, D_train, candidate_alphas,␣
       ↪compute_portfolio_weights, covariance
      ):
          """
          Backtest portfolio optimization strategy using WLS with train-test data.

          Parameters:
          train_panel : DataFrame
              Training panel data.
          test_panel : DataFrame
              Test panel data.
          Y_train : ndarray
              Training target variable (residual returns).
          X_train : ndarray
              Training alpha factors.
          D_train : ndarray
              Training specific risk diagonal.
          candidate_alphas : list
              List of alpha factors to use.
          compute_portfolio_weights : function
              Function to compute portfolio weights.
          covariance : dict
              Dictionary containing factor covariance data by date.

          Returns:
          results : dict
              Dictionary containing cumulative profit, daily profits, risks, market␣
       ↪values, and optimized kappa.
          """
          # Step 1: Cross-validation to find the best kappa
          kappa_values = np.logspace(-6, -2, 10)  # Range of kappa values to test
          best_kappa = cross_validate_wls_with_plot(X_train, Y_train, D_train,␣
       ↪kappa_values)
          print(f"Best Kappa from Cross-Validation: {best_kappa}")

          # Step 2: Train the WLS model on the training data
          coefficients = regularized_wls_train(X_train, Y_train, D_train, best_kappa)

          # Step 3: Backtest on the test panel
          cumulative_profit = 0
          daily_profits = []
          cumulative_profits = []
          daily_risks = []
          idiosyncratic_risks = []
          idiosyncratic_risk_percentages = []
          long_market_values = []
```

```
    short_market_values = []

    grouped = test_panel.groupby(level="Date")

    for date, df in grouped:
        # Compute dynamic risk exposures and factor covariance diagonal
        rske = risk_exposures(df)  # Compute risk exposure matrix (X)
        F_diag = diagonal_factor_cov(date, rske)  # Compute factor covariance␣
↪diagonal using covariance

        # Extract necessary data
        R = df["Y"].to_numpy()  # Residual returns
        X_full = np.asarray(rske)  # Full risk exposure matrix
        D_diag = np.asarray((df["SpecRisk"] / (100 * np.sqrt(252))) ** 2)  #␣
↪Specific risk diagonal

        # Generate alphas (predicted returns) from the trained WLS model
        X = df[candidate_alphas].to_numpy()  # Extract alpha features
        mu = predict_wls(X, coefficients)  # Predicted returns

        # Compute portfolio weights using the Woodbury formula
        weights = compute_portfolio_weights(X_full, F_diag, D_diag, mu)

        # Normalize weights for market value calculations
        total_exposure = np.sum(np.abs(weights))
        normalized_weights = weights / total_exposure

        # Compute daily profit (pre-tcost)
        daily_profit = np.dot(weights, R)
        daily_profits.append(daily_profit)

        # Update cumulative profit
        cumulative_profit += daily_profit
        cumulative_profits.append(cumulative_profit)

        # Compute risk metrics
        factor_covariance = X_full @ F_diag @ X_full.T  # Corrected covariance␣
↪computation
        total_covariance = np.diag(D_diag) + factor_covariance
        portfolio_risk = np.sqrt(weights.T @ total_covariance @ weights)
        idiosyncratic_risk = np.sqrt(weights.T @ np.diag(D_diag) @ weights)
        idiosyncratic_risk_percentage = (idiosyncratic_risk / portfolio_risk) *␣
↪100

        # Store risks
        daily_risks.append(portfolio_risk)
        idiosyncratic_risks.append(idiosyncratic_risk)
```

```
        idiosyncratic_risk_percentages.append(idiosyncratic_risk_percentage)

        # Compute long and short market values
        long_value = np.sum(normalized_weights[normalized_weights > 0])
        short_value = np.sum(np.abs(normalized_weights[normalized_weights < 0]))
        long_market_values.append(long_value)
        short_market_values.append(short_value)

    # Extract the dates for plotting
    dates = [date for date, _ in grouped]

    return {
        "daily_profits": daily_profits,
        "cumulative_profits": cumulative_profits,
        "daily_risks": daily_risks,
        "idiosyncratic_risks": idiosyncratic_risks,
        "idiosyncratic_risk_percentages": idiosyncratic_risk_percentages,
        "long_market_values": long_market_values,
        "short_market_values": short_market_values,
        "best_kappa": best_kappa,
        "dates": dates,
    }
```

[18]:

```
# Run the backtest using the provided inputs
results_train_test = backtest_with_train_test(
    train_panel=train_panel,
    test_panel=test_panel,
    Y_train=Y_train,
    X_train=X_train,
    D_train=D_train,
    candidate_alphas=candidate_alphas,
    compute_portfolio_weights=compute_portfolio_weights,
    covariance=covariance,
)
```

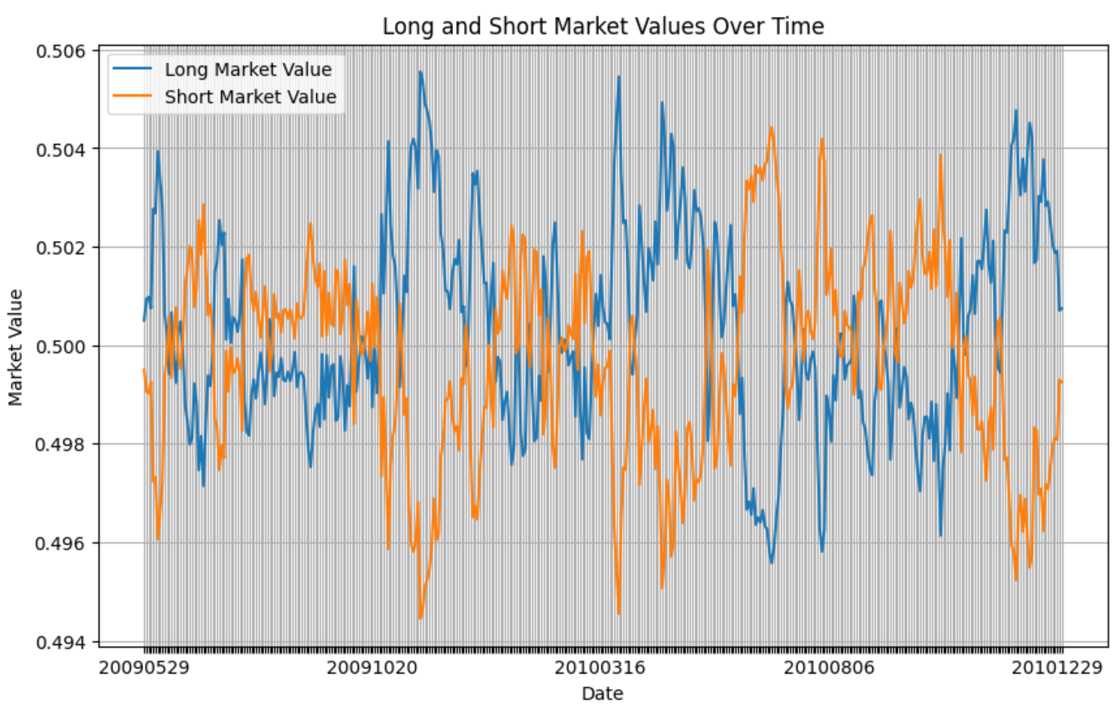


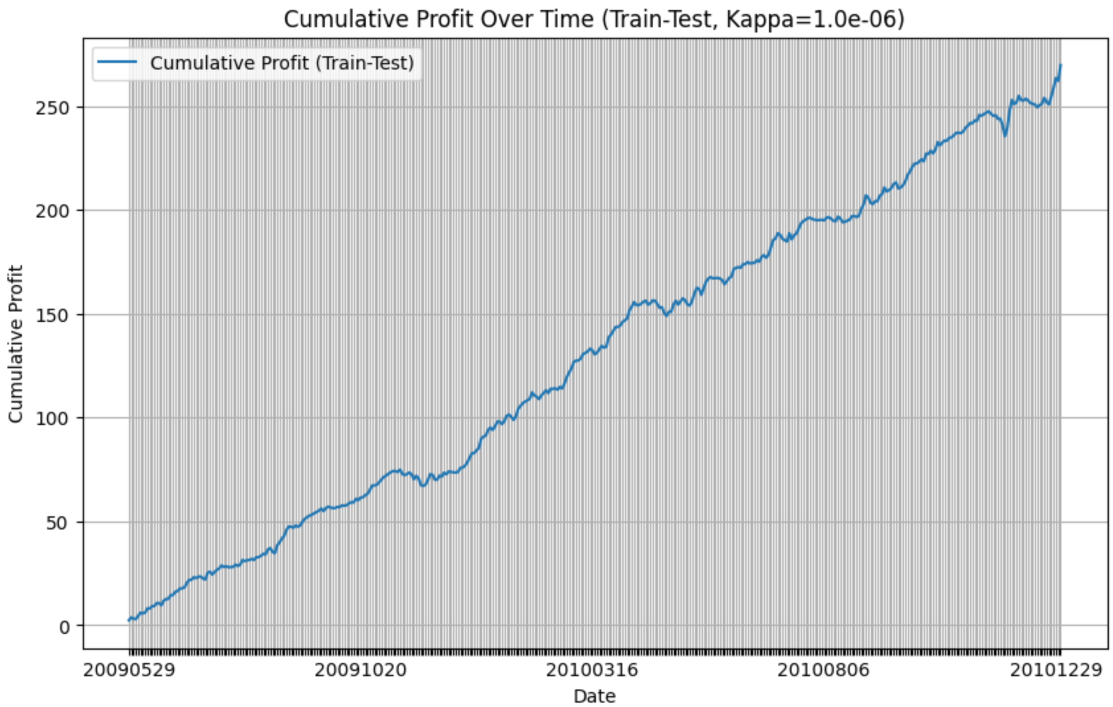

```
Best Kappa from Cross-Validation: 1e-06
```

[19]:
```
# Extract results for plotting
cumulative_profits_train_test = results_train_test["cumulative_profits"]
daily_profits_train_test = results_train_test["daily_profits"]
daily_risks = results_train_test["daily_risks"]
idiosyncratic_risks = results_train_test["idiosyncratic_risks"]
idiosyncratic_risk_percentages =␣
↪results_train_test["idiosyncratic_risk_percentages"]
long_market_values = results_train_test["long_market_values"]
short_market_values = results_train_test["short_market_values"]
best_kappa_train_test = results_train_test["best_kappa"]
dates_train_test = results_train_test["dates"]

# Plot Long and Short Market Values
plt.figure(figsize=(10, 6))
plt.plot(dates_train_test, long_market_values, label="Long Market Value")
plt.plot(dates_train_test, short_market_values, label="Short Market Value")
plt.xlabel("Date")
plt.ylabel("Market Value")
plt.title("Long and Short Market Values Over Time")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
```

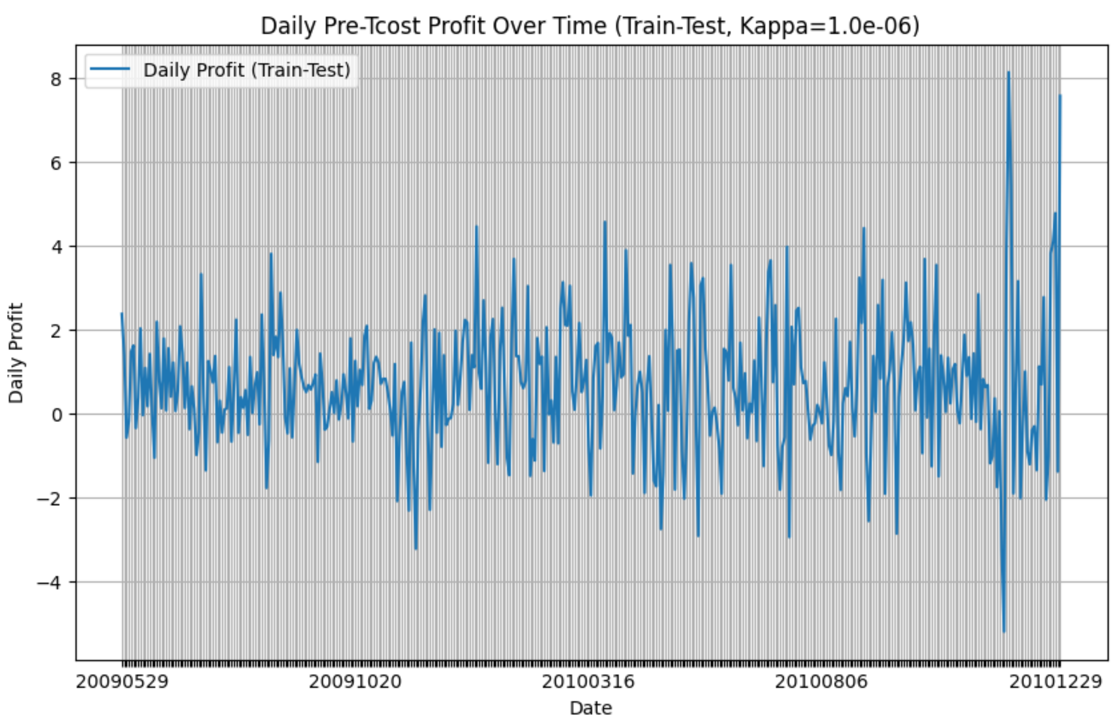
```python
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Cumulative Profit
plt.figure(figsize=(10, 6))
plt.plot(dates_train_test, cumulative_profits_train_test, label="Cumulative␣
 ↪Profit (Train-Test)")
plt.xlabel("Date")
plt.ylabel("Cumulative Profit")
plt.title(f"Cumulative Profit Over Time (Train-Test,␣
 ↪Kappa={best_kappa_train_test:.1e})")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Daily Profits
plt.figure(figsize=(10, 6))
plt.plot(dates_train_test, daily_profits_train_test, label="Daily Profit␣
 ↪(Train-Test)")
plt.xlabel("Date")
plt.ylabel("Daily Profit")
plt.title(f"Daily Pre-Tcost Profit Over Time (Train-Test,␣
 ↪Kappa={best_kappa_train_test:.1e})")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Daily Risks
plt.figure(figsize=(10, 6))
plt.plot(dates_train_test, daily_risks, label="Daily Total Risk")
plt.plot(dates_train_test, idiosyncratic_risks, label="Idiosyncratic Risk")
plt.xlabel("Date")
plt.ylabel("Risk")
plt.title("Daily Portfolio Risks Over Time")
```

```python
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Idiosyncratic Risk Percentage
plt.figure(figsize=(10, 6))
plt.plot(dates_train_test, idiosyncratic_risk_percentages, label="Idiosyncratic␣
 ↪Risk Percentage")
plt.xlabel("Date")
plt.ylabel("Idiosyncratic Risk (%)")
plt.title("Daily Idiosyncratic Risk Percentage Over Time")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()
```

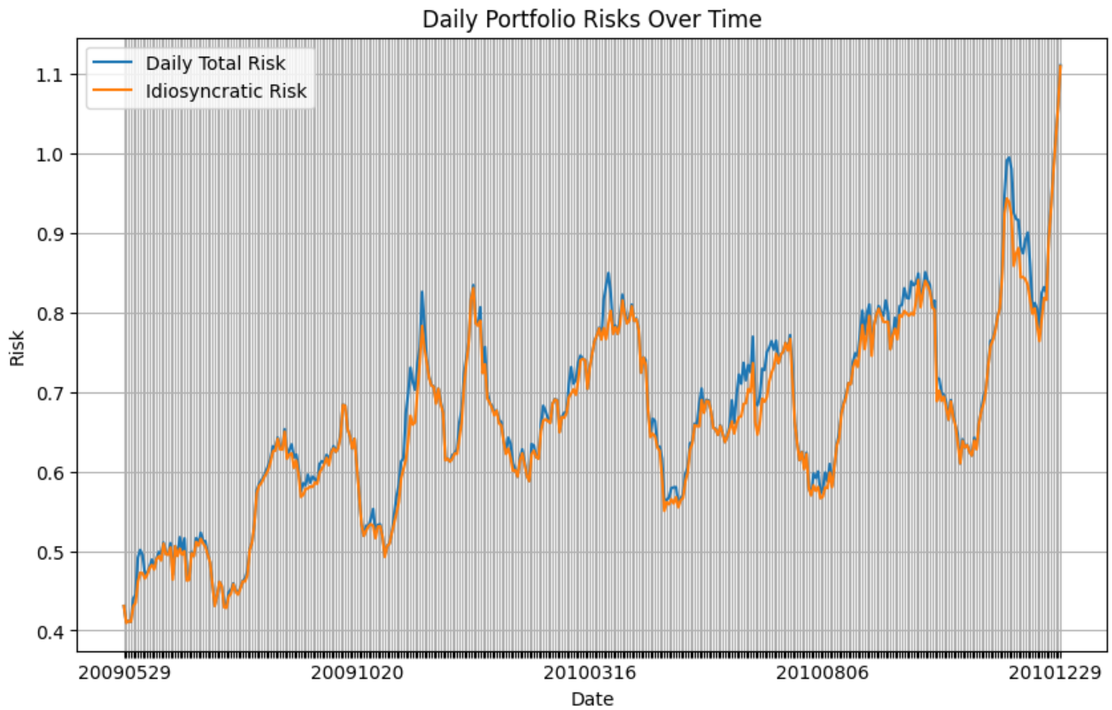


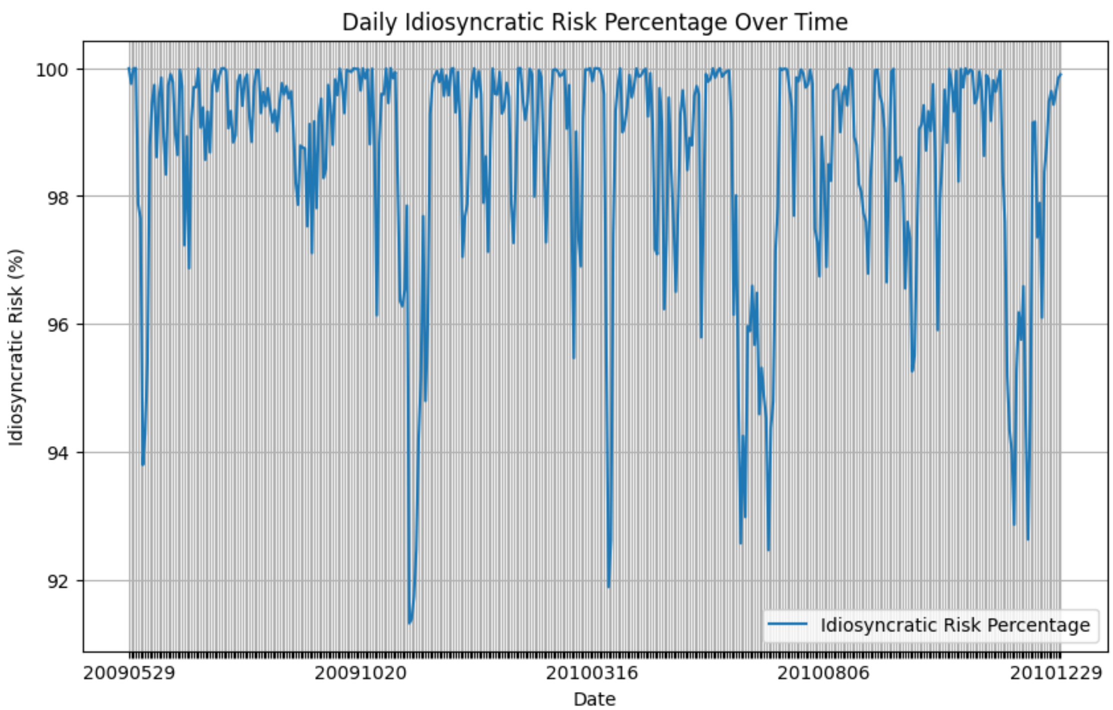


Cumulative Profit Over Time (Train-Test, Kappa=1.0e-06)
Cumulative Profit (Train-Test)
Cumulative Profit
0
50
100
150
200
250
20090529
20091020
20100316
20100806
20101229
Date



Daily Pre-Tcost Profit Over Time (Train-Test, Kappa=1.0e-06)
Daily Profit (Train-Test)
Daily Profit
−4
−2
0
2
4
6
8
20090529
20091020
20100316
20100806
20101229
Date



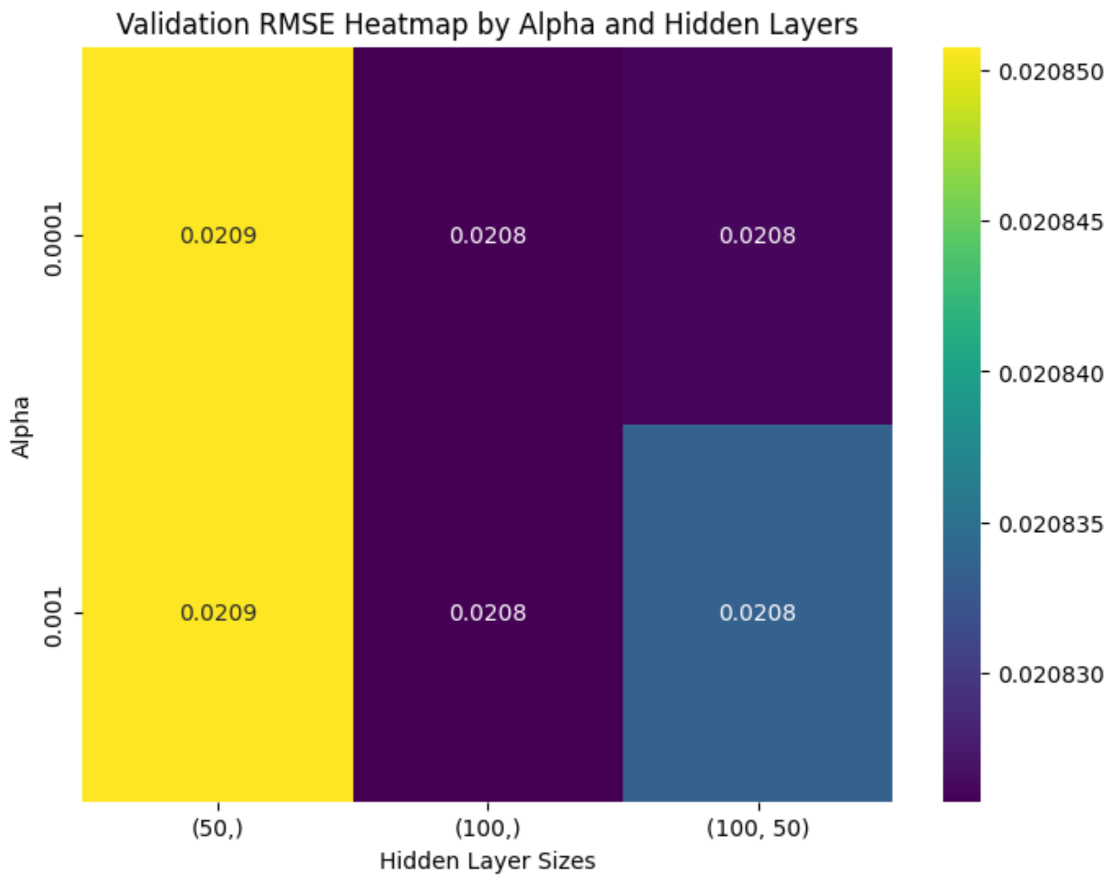


Neural Network

[20]:
```python
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Define the parameter grid for the neural network
param_grid = {
    "hidden_layer_sizes": [(50,), (100,), (100, 50)],  # Different architectures
    "alpha": [0.0001, 0.001],  # Regularization strength
    "learning_rate_init": [0.001],  # Learning rate
}

# Initialize MLP Regressor with reduced max_iter
mlp = MLPRegressor(max_iter=200, random_state=42)  # Set random state for␣
 ↪reproducibility

# Use GridSearchCV to find the best hyperparameters
grid_search = GridSearchCV(
    estimator=mlp,
    param_grid=param_grid,
    scoring="neg_mean_squared_error",
    cv=3,  # 3-fold cross-validation
    n_jobs=-1,  # Use all available CPU cores
    verbose=1,  # Detailed output during the grid search
)

# Fit the model using training data
grid_search.fit(X_train, Y_train)

# Get best parameters and the corresponding model
best_params = grid_search.best_params_
nn_model = grid_search.best_estimator_

# Print best parameters and validation RMSE
print(f"Best Parameters: {best_params}")
val_rmse = np.sqrt(-grid_search.best_score_)
print(f"Validation RMSE: {val_rmse}")

# Evaluate the model on the test set
Y_test_pred = nn_model.predict(X_test)
test_rmse = np.sqrt(mean_squared_error(Y_test, Y_test_pred))
print(f"Test RMSE: {test_rmse}")

# Extract grid search results for visualization
```

```
results = grid_search.cv_results_
hidden_layers = results["param_hidden_layer_sizes"].data
alphas = results["param_alpha"].data
mean_rmse = np.sqrt(-results["mean_test_score"])  # Convert negative MSE to RMSE

# Create DataFrame for heatmap visualization
df_results = pd.DataFrame({
    "Hidden Layers": hidden_layers,
    "Alpha": alphas,
    "Validation RMSE": mean_rmse,
})

# Pivot the DataFrame to plot a heatmap
pivot_table = df_results.pivot_table(index="Alpha", columns="Hidden Layers",␣
 ↪values="Validation RMSE")

# Plot the heatmap for performance
plt.figure(figsize=(8, 6))
sns.heatmap(pivot_table, annot=True, fmt=".4f", cmap="viridis")
plt.title("Validation RMSE Heatmap by Alpha and Hidden Layers")
plt.xlabel("Hidden Layer Sizes")
plt.ylabel("Alpha")
plt.show()
```

```
Fitting 3 folds for each of 6 candidates, totalling 18 fits
Best Parameters: {'alpha': 0.0001, 'hidden_layer_sizes': (100,),
'learning_rate_init': 0.001}
Validation RMSE: 0.020825737724485322
Test RMSE: 0.019315238151596737
```



[26]:
```
def backtest_with_nn_model(
    test_panel,
    nn_model,
    candidate_alphas,
    compute_portfolio_weights,
    covariance
):
    """
    Backtest portfolio optimization strategy using a pretrained neural network␣
↪model.

    Parameters:
    test_panel : DataFrame
        Test panel data.
    nn_model : sklearn.neural_network.MLPRegressor
        Pretrained neural network model.
    candidate_alphas : list
        List of alpha factors to use.
    compute_portfolio_weights : function
```

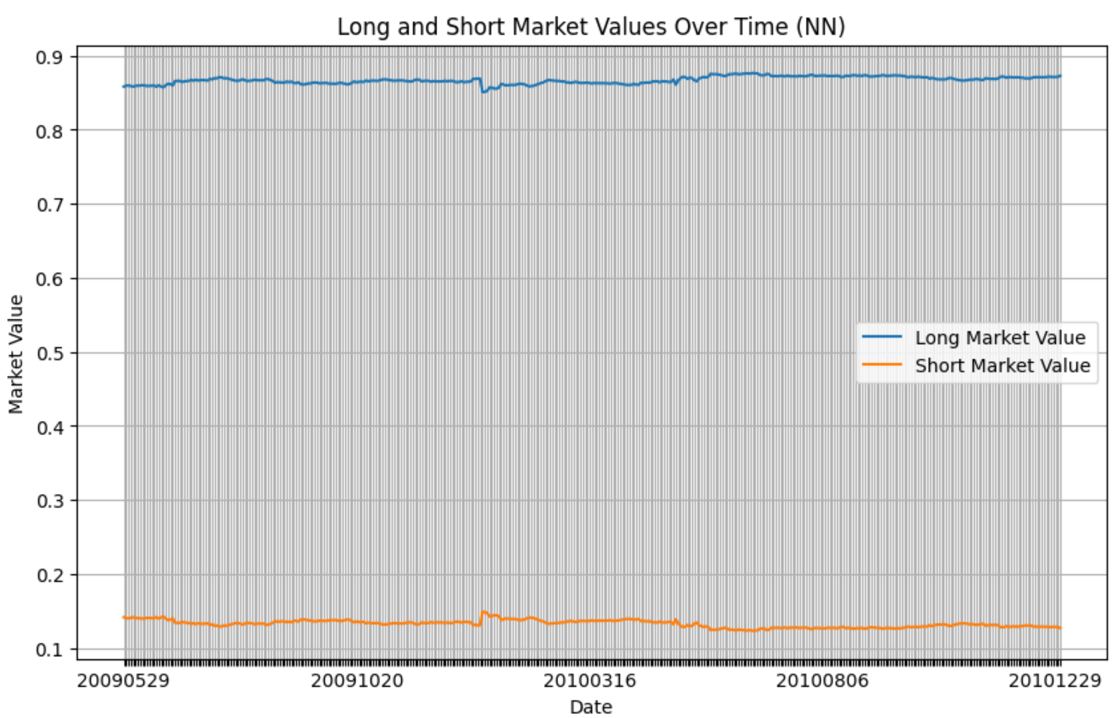
```python
        Function to compute portfolio weights.
    covariance : dict
        Dictionary containing factor covariance data by date.

    Returns:
    results : dict
        Dictionary containing cumulative profit, daily profits, risks, and␣
↪market values.
    """
    cumulative_profit = 0
    daily_profits = []
    cumulative_profits = []
    daily_risks = []
    idiosyncratic_risks = []
    idiosyncratic_risk_percentages = []
    long_market_values = []
    short_market_values = []

    grouped = test_panel.groupby(level="Date")

    for date, df in grouped:
        # Compute risk exposures and factor covariance diagonal
        rske = risk_exposures(df)  # Risk exposure matrix
        F_diag = diagonal_factor_cov(date, rske)  # Factor covariance diagonal
        X_full = np.asarray(rske)  # Full risk exposure matrix
        D_diag = np.asarray((df["SpecRisk"] / (100 * np.sqrt(252))) ** 2)  #␣
↪Specific risk diagonal

        # Predict expected returns using the pretrained neural network
        X = df[candidate_alphas].to_numpy()  # Extract alpha features
        mu = np.abs(nn_model.predict(X))  # Predicted returns

        # Compute portfolio weights using the Woodbury formula
        weights = compute_portfolio_weights(X_full, F_diag, D_diag, mu)

        # Normalize weights for market value calculations
        total_exposure = np.sum(np.abs(weights))
        normalized_weights = weights / total_exposure

        # Compute daily profit (pre-tcost)
        R = df["Y"].to_numpy()  # Residual returns
        daily_profit = np.dot(weights, R)
        daily_profits.append(daily_profit)

        # Update cumulative profit
        cumulative_profit += daily_profit
        cumulative_profits.append(cumulative_profit)
```

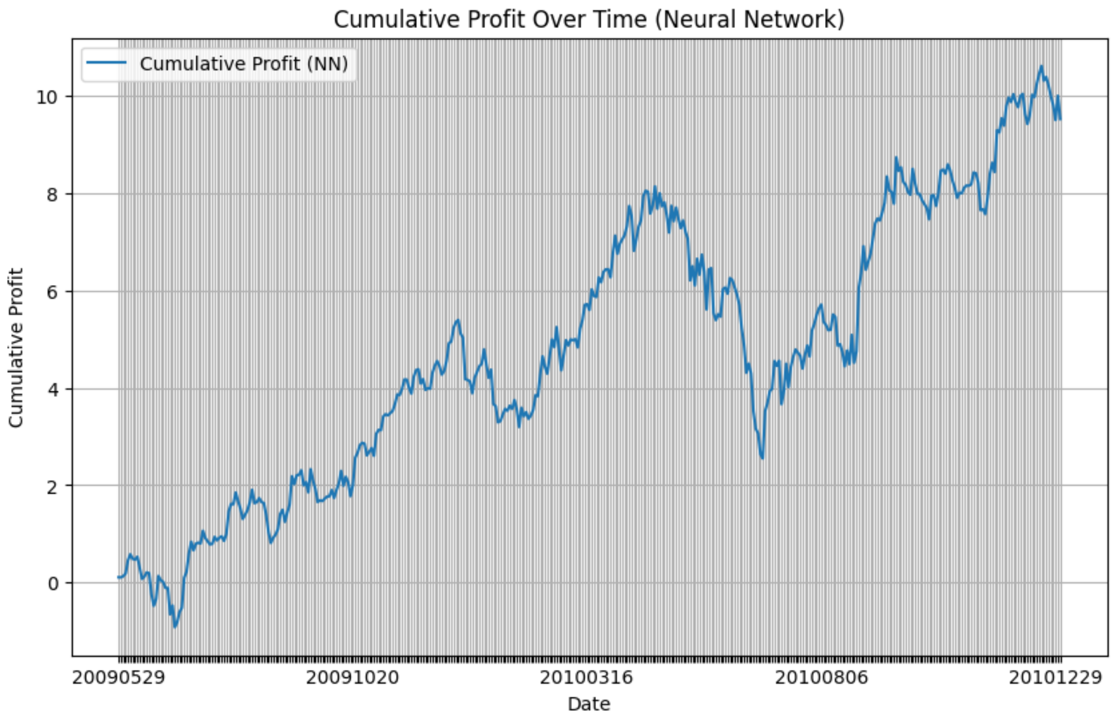
```
        # Compute risk metrics
        factor_covariance = X_full @ F_diag @ X_full.T
        total_covariance = np.diag(D_diag) + factor_covariance
        portfolio_risk = np.sqrt(weights.T @ total_covariance @ weights)
        idiosyncratic_risk = np.sqrt(weights.T @ np.diag(D_diag) @ weights)
        idiosyncratic_risk_percentage = (idiosyncratic_risk / portfolio_risk) *␣
↪100

        # Compute long and short market values
        long_value = np.sum(normalized_weights[normalized_weights > 0])
        short_value = np.sum(np.abs(normalized_weights[normalized_weights < 0]))

        # Store metrics
        daily_risks.append(portfolio_risk)
        idiosyncratic_risks.append(idiosyncratic_risk)
        idiosyncratic_risk_percentages.append(idiosyncratic_risk_percentage)
        long_market_values.append(long_value)
        short_market_values.append(short_value)

    # Extract the dates for plotting
    dates = [date for date, _ in grouped]

    return {
        "daily_profits": daily_profits,
        "cumulative_profits": cumulative_profits,
        "daily_risks": daily_risks,
        "idiosyncratic_risks": idiosyncratic_risks,
        "idiosyncratic_risk_percentages": idiosyncratic_risk_percentages,
        "long_market_values": long_market_values,
        "short_market_values": short_market_values,
        "dates": dates,
    }


# Run the backtest using the pretrained neural network
results_nn_backtest = backtest_with_nn_model(
    test_panel=test_panel,
    nn_model=nn_model,  # Pretrained neural network model
    candidate_alphas=candidate_alphas,
    compute_portfolio_weights=compute_portfolio_weights,
    covariance=covariance,
)
```

[27]:
```
# Extract results for plotting
cumulative_profits_nn = results_nn_backtest["cumulative_profits"]
daily_profits_nn = results_nn_backtest["daily_profits"]
```

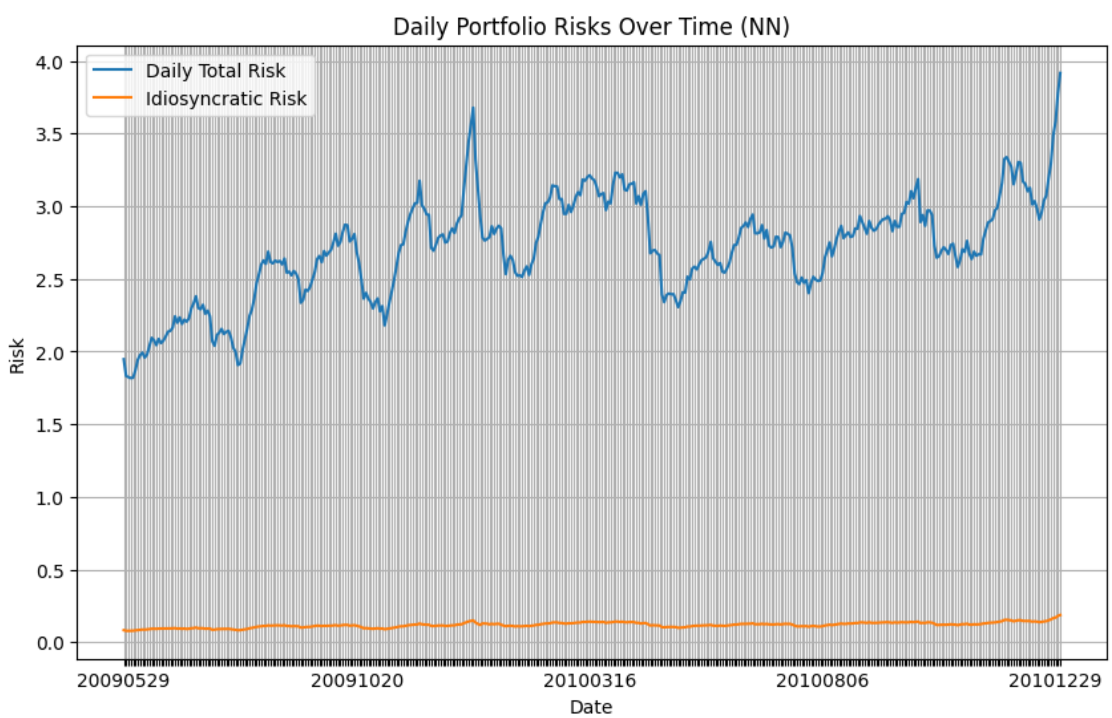
```
daily_risks_nn = results_nn_backtest["daily_risks"]
idiosyncratic_risks_nn = results_nn_backtest["idiosyncratic_risks"]
idiosyncratic_risk_percentages_nn =␣
 ↪results_nn_backtest["idiosyncratic_risk_percentages"]
long_market_values_nn = results_nn_backtest["long_market_values"]
short_market_values_nn = results_nn_backtest["short_market_values"]
dates_nn = results_nn_backtest["dates"]

# Visualization
# Plot Long and Short Market Values
plt.figure(figsize=(10, 6))
plt.plot(dates_nn, long_market_values_nn, label="Long Market Value")
plt.plot(dates_nn, short_market_values_nn, label="Short Market Value")
plt.xlabel("Date")
plt.ylabel("Market Value")
plt.title("Long and Short Market Values Over Time (NN)")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Cumulative Profit
plt.figure(figsize=(10, 6))
plt.plot(dates_nn, cumulative_profits_nn, label="Cumulative Profit (NN)")
plt.xlabel("Date")
plt.ylabel("Cumulative Profit")
plt.title("Cumulative Profit Over Time (Neural Network)")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Daily Risks
plt.figure(figsize=(10, 6))
plt.plot(dates_nn, daily_risks_nn, label="Daily Total Risk")
plt.plot(dates_nn, idiosyncratic_risks_nn, label="Idiosyncratic Risk")
plt.xlabel("Date")
plt.ylabel("Risk")
plt.title("Daily Portfolio Risks Over Time (NN)")
```

```
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()

# Plot Idiosyncratic Risk Percentage
plt.figure(figsize=(10, 6))
plt.plot(dates_nn, idiosyncratic_risk_percentages_nn, label="Idiosyncratic Risk␣
 ↪Percentage")
plt.xlabel("Date")
plt.ylabel("Idiosyncratic Risk (%)")
plt.title("Daily Idiosyncratic Risk Percentage Over Time (NN)")
plt.legend()
for ind, label in enumerate(plt.gca().get_xticklabels()):
    if ind % 100 == 0:
        label.set_visible(True)
    else:
        label.set_visible(False)
plt.grid()
plt.show()
```

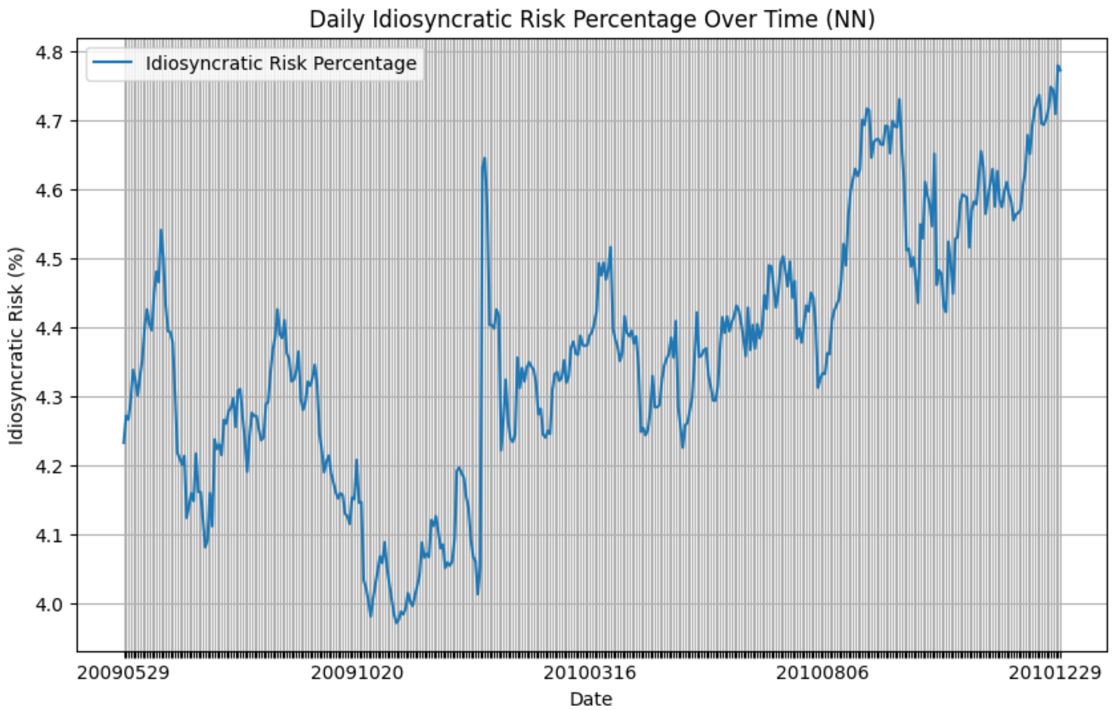




Cumulative Profit Over Time (Neural Network)
Cumulative Profit (NN)
Cumulative Profit
10
8
6
4
2
0
20090529
20091020
20100316
20100806
20101229
Date



Daily Portfolio Risks Over Time (NN)
Daily Total Risk
Idiosyncratic Risk
Risk
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
20090529
20091020
20100316
20100806
20101229
Date


Daily Idiosyncratic Risk Percentage Over Time (NN)
Idiosyncratic Risk Percentage
Idiosyncratic Risk (%)
4.8
4.7
4.6
4.5
4.4
4.3
4.2
4.1
4.0
20090529
20091020
20100316
20100806
20101229
Date

[ ]: